MASTER EN SCIENCES INFORMATIQUES

Réalisation d'un programme de vérification des invariants

Simon, Luc

Award date:
1992

Awarding institution:
Universite de Namur

REALISATION D'UN PROGRAMME
DE VERIFICATION
DES INVARIANTS

Luc SIMON

Mémoire réalisé sous la
direction du Professeur B. Le Charlier
en vue de l'obtention du titre de
Licencié et Maître en Informatique.

Namur Septembre 1992

Bref résumé de ce mémoire

La nécessité d'arriver à une meilleure qualité des programmes est actuellement reconnue, et des méthodes de construction de programme sont développées à cet effet. Ainsi, aux Facultés Universitaires de Namur, une telle méthode est enseignée : la méthode de l'invariant. Ce mémoire réalise un programme d'aide à l'enseignement de la programmation basé sur cette méthode. La construction d'un programme est basée sur la description des situations de ce programme. Une situation est l'expression de conditions sur l'état des variables et tableaux du programme à un moment déterminé de son exécution. Ce mémoire définit un langage permettant de définir les conditions et les instructions du programme. Une extension graphique est possible et définie par Jeannine Rulkin. Ce mémoire réalise également un programme qui vérifie les situations définies pour un programme.

Abstract

The correctness of program construction is currently recognized as a major issue, and appropriate methodologies are therefore developped. Such a methodology, called the "Invariant Method", is teached at the Institut d'Informatique of the Facultés Universitaires de Namur. This thesis realises an interactive program and designed to help the students to learn the metodology. The porcess of program construction is based on a description of situations of this program. A situation expresses conditions on the program's variables and arrays state, at a given stage of its execution. This thesis defines a language allowing the expression of these conditions and the instructions of the program. A graphical representation is defined by Jeannine Rulkin. This thesis realises also a program that verifies the situations of a program.

Table des matières

Remerciements

Que toutes les personnes m'ayant aidé au cours de cette dernière année d'études trouvent ici l'expression de mes remerciements.

Je remercie particulièrement Baudouin Le Charlier, promoteur de ce mémoire, pour ses explications et ses conseils tout au long de la réalisation de ce travail. Je remercie mes parents pour l'aide lors de la mise en page du texte final.

Introduction

Nombreuses sont les personnes qui pensent que, pour programmer, il suffit de connaître un langage de programmation et qu'il suffit de placer des instructions les unes à la suite des autres.

En général, le premier langage que l'on connaît est le Basic. On écrit donc quelques instructions, on les essaie sur quelques données et, si le résultat obtenu n'est pas correct, on modifie les instruction et on recommence. Cela se résume à construire un programme avec des "bouts de programmes" qui fonctionnent correctement avec les données qui ont permis de le tester.

Cette méthode de construction demande beaucoup de temps, car on procède par "essais et erreurs". Avec les langages procéduraux comme le PASCAL et le C, il n'est plus aussi facile de vérifier une partie de programme : il faut construire TOUT un programme pour tester UNE partie.

Il est donc intéressant de posséder une méthode qui permette de construire un programme[1] et de le vérifier au fil de sa construction. Une méthode de travail est enseignée aux FACULTES NOTRE-DAME DE LA PAIX : il s'agit de la méthode des invariants. Ces invariants sont des conditions sur les données manipulées qui ne tiennent pas comptes de valeurs particulières mais de conditions plus générales.

Il y a trois groupe de conditions :

- les conditions qui doivent être vérifiées par les variables avant d'exécuter le programme, c'est-à-dire la précondition du programme;
- celles qui doivent correspondent à la fin du programme, c'est-à-dire la postcondition. Elle doit exprimer le but du programme;
- et enfin les conditions pour les boucles, c'est-à-dire les invariants.

Pour éclairer ces propos, nous citerons quelques exemples de conditions.
Ainsi :

a) les i premiers éléments du tableau ont été examinés;

b) les i premiers éléments d'un tableau sont tous plus petits ou égaux à la valeur contenue dans la variable max;

c) si q et r sont respectivement le quotient et le reste de la division entière de a par b, on doit avoir a = q * b + r.

---

1 Dans cette introduction, 'programme' signifie, soit un programme en entier, soit une partie d'un programme complet.

Ces conditions ne tiennent pas compte des valeurs contenues dans les variables[2] du programme.

S'il est prouvé que le programme est correct grâce aux conditions posées sur les données, il sera également correct dans tous les cas où les données correspondent aux critères. Dans le cas contraire, aucune affirmation ne peut être faites.

Cette méthode, basée sur des formulations mathématiques, possède un inconvénient majeur : elle est utilisable pour de petits programmes (recherche du minimum d'un tableau, calcul de la somme des éléments d'un tableau, ...). Lorsque le programme devient plus complexe, cette méthode demande trop d'écritures pour vérifier l'exactitude du programme. Malgré cet inconvénient, elle permet de donner au programmeur une "tournure d'esprit dans la conception des programmes" qu'il utilisera le plus souvent inconsciemment.

Le but recherché par ce travail est de construire un programme qui permettrait d'apporte de l'aide aux étudiants des Facultés lors des exercices dans le cadre du cours de Programmation. Il devrait vérifier que les conditions posées par les étudiants soient bien formulées et que les instructions du programme aboutissent à la postcondition souhaitée.

Deux mémoires ont déjà participé à cette recherche :

- D. Fisette, en 1985, a défini un langage de programmation permettant l'expression d'assertions;
- Jeannine Rulkin, en 1989, a défini le langage de conditions graphiques.

Ce mémoire est la réalisation d'un noyau de base permettant de définir des conditions sur les variables, et des instructions manipulant ces variables. C'est à Jeannine RULKIN que nous empruntons les définitions des variables et des fonctions disponibles pour les conditions.

---

2 Par 'variables', j'inclus tout ce que l'on peut déclarer dans un langage : variables simples, tableaux, fichiers, ...

Chapitre I. Rappel

Ce chapitre est repris du mémoire de Jeannine Rulkin.

1. Définition des types de valeurs

Dans le langage de condition, deux types de valeurs sont permis:

1.1. Le type ENTIER

L'ensemble des objets de type entier : Z = {-32500..32500}

Les opérations primitives

Addition : Z x Z -> Z : (a,b) --> a + b
Soustraction : Z x Z -> Z : (a,b) --> a - b
Multiplication : Z x Z -> Z : (a,b) --> a * b
Division entière : Z x Z -> Z : (a,b) --> a div b (b <> 0)
Reste : Z x Z -> Z : (a,b) --> a mod b (b <> 0)
Egalité : Z x Z -> B : (a,b) --> a = b
Inégalité : Z x Z -> B : (a,b) --> a <> b

(de même pour les autres types de comparaison : >, >=, <, <=)

1.2. Le type BOOLEEN

L'ensemble des objets de type booléen : B = {vrai,faux}

La valeur 1 est associée à vrai tandis que 0 est associée à faux. Cette convention est utilisée par le programme pour afficher et lire des données.

Les opérations primitives

Conjonction : B x B -> B : (a,b) --> a and b
Disjonction : B x B -> B : (a,b) --> a or b
Négation : B --> B : a --> not a

2. Quelques définitions

Environnement

L'environnement d'un programme est l'ensemble des constantes, variables, tableaux, segments et expressions utilisés dans le programme.

Avant l'exécution du programme, les variables et tableaux sont initialisés et une copie de ces valeurs est effectuée. Cette copie ne sera jamais modifiée et s'appelle l'**environnement initial** du programme.

Mémoires, variables et tableaux

Désignons par "mémoire" tout support physique vérifiant les quatre propriétés suivantes :

- il est possible d'y enregistrer de l'information
- l'information enregistrée peut être lue
- l'information contenue ne change pas tant qu'il n'y a pas de nouvel enregistrement
- une mémoire est caractérisée par le type d'information qu'elle peut contenir (entier ou booléen dans le cadre du langage de conditions restreint que nous définissons).

On appelle **VARIABLE SIMPLE**, l'association d'un identificateur et d'une mémoire. L'identificateur est appelé le ***nom de la variable***, la valeur (si elle existe) dont la représentation est enregistrée dans la mémoire à un instant donné, est la ***valeur courante de la variable.*** On appelle type de la variable le type de sa mémoire.

On appelle **TABLEAU**, l'association d'un identificateur (le ***nom du tableau***) et d'une bijection d'un intervalle fini de nombres entiers (les ***indices du tableau***) dans un ensemble de mémoires (les ***éléments du tableau***) de même type (le ***type du tableau***; le langage de condition restreint n'admettant que des tableaux de type entier).

On appelle **VARIABLE INDICEE**, l'association d'un tableau et d'un indice de ce tableau (dans le langage de condition restreint, une variable indicée doit toujours être de type entier).

Notion de segment

Un **SEGMENT** est constitué d'éléments d'indices successifs d'un tableau.

Un **segment** peut avoir un nom. Il est caractérisé par :

- une borne inférieure, associée à l'indice du premier élément du segment dans le tableau
- une borne supérieure, associée à l'indice du dernier élément du segment dans le tableau.

Il est possible de définir un segment grâce au nom du tableau et à deux expressions qui limitent les indices des éléments. Cette notion de segment est utile pour les expressions "pour tout" et "il existe".

Expression de désignation

Une expression de désignation de variable (de tableau, ou de segment) est une construction du langage qui, étant donné un environnement E, identifie une et une seule variable (tableau, ou segment).

Syntaxe :

‹nom de variable› ::= ‹identificateur›

‹nom de tableau› ::= ‹identificateur›

‹nom de segment› ::= ‹identificateur›

‹expression de désignation de tableau› ::= ‹nom de tableau›

‹expression de désignation de variable› ::=
‹nom de variable› |
‹expression de désignation de tableau› [‹expression arithmétique›]

‹borne› ::= ‹expression arithmétique›

‹expression de désignation de segment› ::=
‹expression de désignation de tableau› { ‹borne› : ‹borne› } |
‹nom de segment›

Sémantique

Une expression de désignation possède un type.

Le type d'une expression de désignation de tableau (segment) est le type du tableau (segment) désigné; le langage

de conditions restreint n'admet que les tableaux (segments) de type entier.

Une borne est une expression arithmétique de type entier.

Une expression de désignation de variable est du type de la variable désignée, c'est-à-dire soit de type entier soit de type booléen, cependant, une expression de désignation de variable indicée est de type entier (car désigne un élément de tableau).

Sémantique d'une expression de désignation de tableau :

Soient :

- E, un environnement,
- t, un identificateur,

si t est un nom de tableau dans E alors l'expression de désignation t désigne ce tableau.

Sémantique d'une expression de désignation de variable :

Soient :

- E, un environnement
- x et t, des identificateurs

si x est un nom de variable dans E alors l'expression de désignation x désigne cette variable,

si t est un nom de tableau dans E alors l'expression de désignation t[expr], où expr est une expression arithmétique, désigne une variable indicée évaluée de la manière suivante :

1- on évalue l'expression arithmétique expr dans E,

2- si l'évaluation de expr est indéterminée alors l'évaluation de t[expr] est indéterminée, sinon soit i, la valeur obtenue lors de l'évaluation de expr dans E, si i est un indice du tableau t, alors l'expression de désignation t[expr] désigne l'unique variable indicée t[i] sinon l'évaluation de t[i] est indéterminée.

Sémantique d'une expression de désignation de segment :

Soient

- E, un environnement,
- t et s, des identificateurs,

- bi et bs, des bornes (ou expressions arithmétiques),

si s est un nom de segment dans E alors l'expression de désignation s désigne ce segment,

si t est un nom de tableau dans E alors l'expression de désignation t[bi:bs] désigne un segment du tableau t qui est caractérisé par les bornes bi et bs, bi étant la borne inférieure et bs la borne supérieure.

Expression

Syntaxe

```
<expression> ::= <expression arithmétique> |
                 <expression booléenne>
```

Syntaxe des expressions arithmétiques

```
<opérateur multiplicatif> ::= * | div | mod

<opérateur additif> ::= + | -

<facteur> ::= <entier> |
              <expression de désignation de variable> |
              <appel de fonction de type entier> |
              ( <expression arithmétique> )

<terme> ::= <facteur> |
            <terme> <opérateur multiplicatif> <facteur>

<expression arithmétique> ::= <terme> |
                    -<terme> |
                    <expression arithmétique> <opérateur additif>
                          <terme>
```

Syntaxe des expressions booléennes

```
<opérateur relationnel> ::= = | <= | >= | <> | < | >

<expression booléenne simple> ::=
                    <expression de désignation de type booléen> |
                    <valeur de vérité> |
                    <appel de fonction de type booléen> |
                    ( <expression booléenne> )

<proposition atomique> ::= <expression booléenne simple> |
                <expression arithmétique> <opérateur relationnel>
                          <expression arithmétique>

<négation> ::= <proposition atomique> |
               not <proposition atomique>
```

<conjonction> ::= <négation> | <négation> **and** <conjonction>

<expression booléenne> ::= <conjonction> |
<conjonction> **or** <expression booléenne>

Evaluation d'une expression arithmétique élémentaire

Soient :

- E, un environnement,
- term, un terme,
- expr, une expression arithmétique élémentaire,
- w1, un opérateur multiplicatif,
- w2, un opérateur additif
- fact, un facteur.

Dans l'environnement E, les expressions ci-dessous sont évaluées de la manière suivante :

fact :

Si **fact** est un entier, alors l'évaluation de fact donne comme valeur cet entier (l'évaluation d'une constante produit toujours la même valeur quelle que soit l'environnement).

Si **fact** est une expression de désignation de variable (c'est-à-dire une variable ou une variable indicée de type entier), alors l'évaluation de fact donne comme valeur la valeur de la variable désignée par l'expression de désignation dans l'environnement E.

Si **fact** est un appel de fonction de type entier alors cfr la sémantique et la valeur de la fonction en question (les seules fonctions admises étant les fonctions prédéfinies).

Si **fact** est une expression entre parenthèses alors l'évaluation de fact est l'évaluation de cette expression.

term w1 fact :

On effectue l'évaluation de **term**, puis l'évaluation de **fact**, dans l'environnement E.

Soient v1 et v2, les valeurs obtenues, si elles ne sont pas de type entier alors la valeur de l'expression **term w1 fact** n'existe pas, si elles sont de type entier alors la

valeur de l'expression **term w1 fact** est v1 w1 v2 (c'est-à-dire la valeur de l'opération primitive w1 appliquée à v1 et v2).

expr w2 term :

On effectue l'évaluation de **expr**, puis l'évaluation de **term**, dans l'environnement E.

Soient v1 et v2, les valeurs obtenues, si elles ne sont pas de type entier alors la valeur de l'expression **expr w2 term** n'existe pas, si elles sont de type entier alors la valeur de l'expression **expr w2 term** est v1 w1 v2 (c'est-à-dire la valeur de l'opération primitive w2 appliquée à v1 et v2).

-term :

On effectue l'évaluation de **term** dans l'environnement E.

Soit v, la valeur obtenue, si elle n'est pas de type entier alors la valeur de l'expression **-term** n'existe pas, si v est de type entier alors la valeur de l'expression **-term** est -v.

L'évaluation d'une expression arithmétique élémentaire a donc pour effet de renvoyer une valeur de type entier.

L'évaluation d'une expression booléenne élémentaire

Soit :

- E, un environnement,
- E_0, l'environnement initial de E,
- simplbexpr, une expression booléenne simple
- aexpr1 et aexpr2, deux expressions booléennes arithmétiques
- aprop, une proposition atomique
- neg, une négation,
- conj, une conjonction,
- bexpr, une expression booléenne,
- w, un opérateur relationnel.

Etant donné l'environnement initial E_0 et l'environnement E, les expressions ci-dessous sont évaluées de la manière suivante :

Fonctions

Soit **SEG** représentant l'ensemble des segments et des tableaux

<suite> ::= <expression de désignation de segment> |
<expression de désignation de tableau>

Soit

- S_1, S_2, S_3, ... des segments ou tableaux
- bi_1, bi_2, bi_3, .. les bornes inférieures des segments S_i
- bs_1, bs_2, bs_3, .. les bornes supérieures des segments S_i

t_1 est un tableau dans lequel on peut trouver le segment S_1

La fonction SOMME : SEG --> Z

somme(<suite>)

La fonction SOMME appliquée au segment[1] S_1 renvoie une valeur v de type entier telle que v est égale à la somme de tous les éléments du segment S_1 :

$$v = S_1[bi_1] + S_1[bi_1 + 1] + \ldots + S_1[bs_1]$$

La fonction PRODUIT : SEG --> Z

produit(<suite>)

La fonction PRODUIT appliquée au segment S_1 renvoie une valeur v de type entier telle que v est égale au produit de tous les éléments du segment S_1 :

$$v = S_1[bi_1] * S_1[bi_1 + 1] * \ldots * S_1[bs_1]$$

1 Dans la description des fonctions, le mot "segment" signifiera soit un tableau, soit un segment.

La fonction MINIMUM (MAXIMUM) : SEG --> Z

min(<suite>)
max(<suite>)

La fonction MINIMUM (MAXIMUM) appliquée au segment S_1 renvoie une valeur v de type entier telle que v est égale à la valeur minimale (maximale) du segment S_1 :

$v = \min (\max)\ S_1[i]$ pour $bi_1 <= \leq i \leq bs_1$

La fonction LONG : SEG --> Z

long(<suite>)

La fonction LONG appliquée au segment S_1 renvoie une valeur v de type entier telle que v est égale au nombre d'éléments du segment S_1 :

$v = bs_1 - bi_1 + 1$ si $bs_1 \geq bi_1$

$v = 0$ sinon

La fonction FIRST (LAST) : SEG --> Z

first(<suite>)
last(<suite>)

La fonction FIRST (LAST) appliquée au segment S_1 renvoie une valeur v de type entier telle que v est égale à la valeur du premier (dernier) élément du segment S_1 :

$v = S_1[bi_1]$
$(v = S_1[bs_1])$

La fonction HEAD (TAIL) : SEG --> SEG

head(<suite>)
tail(<suite>)

La fonction HEAD (TAIL) appliquée au segment S_1 renvoie un segment S' tel que S' est le même segment que S_1 tronqué de son dernier (premier) élément :

$S' = t1\{bi_1 : bs_1 - 1\}$
$(S' = t1\{bi_1 + 1 : bs_1\})$

Le prédicat INCHANGE : SEG --> B

inchange(<suite>)

Le prédicat INCHANGE appliqué au segment S_1 renvoie une valeur v de type booléen telle que

v vaut **vrai** si les valeurs enregistrées dans les mémoires associées aux éléments du segment S_1 dans l'environnement initial E_0 égalent les valeurs enregistrées dans les mémoires associées aux éléments de S_1 dans l'environnement E

et sinon v vaut **faux.**

Le prédicat EMPTY : SEG --> B

vide(<suite>)

Le prédicat VIDE appliqué au segment S_1 renvoie une valeur v de type booléen telle que v vaut **vrai** si le segment S_1 est vide et **faux** sinon.

Le prédicat CONCAT : SEG x SEG x SEG --> B

concat(<suite>, <suite>, <suite>)

Le prédicat CONCAT appliqué aux segments S_1, S_2 et S_3 renvoie une valeur v de type booléen telle que v vaut **vrai** si

$$(S_1[bi_1], \ldots, S_1[bs_1], S_2[bi_2], \ldots, S_2[bs_2]) = (S_3[bi_3], \ldots, S_3[bs_3])$$

Le prédicat EGAL : SEG x SEG --> B

egal(<suite>, <suite>)

Le prédicat EGAL appliqué aux segments S_1 et S_2 renvoie une valeur v de type booléen telle que

v vaut **vrai** si le segment S_1 égale le segment S_2, c'est-à-dire si

$$S_1[bi_1] = S_2[bi_2], \ldots, S_1[bs_1] = S_2[bs_2]$$

et **faux** sinon.

Le prédicat PERMUTATION : SEG x SEG --> B

permutation(<suite>, <suite>)

Le prédicat PERMUTATION appliqué aux segments S_1 et S_2 renvoie une valeur v de type booléen telle que v vaut **vrai** si les éléments du segment S_2 représentent une permutation des éléments du segment S_1, et **faux** sinon.

Le prédicat SEGMENT : SEG x SEG --> B

segment(<suite>, <suite>)

Le prédicat SEGMENT appliqué aux segments S_1 et S_2 renvoie une valeur v de type booléen telle que v vaut **vrai** si le segment S_2 est un sous-segment de S_1 (c'est-à-dire $bi_1 \leq bi_2 \leq bs_2 \leq bs_1$) et **faux** sinon.

Le prédicat PREFIXE (SUFFIXE) : SEG x SEG --> B

prefixe(<suite>, <suite>)
suffixe(<suite>, <suite>)

Le prédicat PREFIXE (SUFFIXE) appliqué aux segments S_1 et S_2 renvoie une valeur v de type booléen telle que v vaut **vrai** si le segment S_2 est un préfixe (suffixe) du segment S_1,

c'est-à-dire $bi_1 = bi_2 \leq bs_2 \leq bs_1$ ($bi_1 \leq bi_2 \leq bs_2 = bs_1$)

et **faux** sinon.

Le prédicat TRICROIS (TRIDEC) : SEG --> B

tricrois(<suite>)
tridec(<suite>)

Le prédicat TRICROIS (TRIDEC) appliqué au segment S_1 renvoie une valeur v de type booléen telle que

v vaut **vrai** si les éléments du segment sont rangés dans le segment par ordre croissant (décroissant), c'est-à-dire

$$t[bi_1] \leq t[bi_1+1] \leq \ldots \leq t[bs_1]$$
$$(t[bi_1] \geq t[bi_1+1] \geq \ldots \geq t[bs_1])$$

et **faux** sinon.

Le prédicat TRISTRCROIS (TRISTRDEC) : SEG --> B

```
tristrcrois( <suite> )
tristrdec( <suite> )
```

Le prédicat TRISTRCROIS (TRISTRDEC) appliqué au segment S_1 renvoie une valeur v de type booléen telle que

v vaut **vrai** si les éléments du segment sont rangés dans le segment par ordre strictement croissant (décroissant), c'est-à-dire

$$t[bi_1] < t[bi_1+1] < \ldots < t[bs_1]$$
$$(t[bi_1] > t[bi_1+1] > \ldots > t[bs_1])$$

et **faux** sinon.

Le prédicat PAIR : SEG --> B

```
pair( <suite> )
```

Le prédicat PAIR appliqué au segment S_1 renvoie une valeur v de type booléen telle que v vaut **vrai** si tous les éléments du segment S_1 sont de valeur paire et **faux** sinon.

Le prédicat EXAMINE : SEG --> B

```
examine( <suite> )
```

Le prédicat EXAMINE appliqué au segment S_1 renvoie une valeur v de type booléen telle que v vaut **vrai** si le segment S_1 a été examiné et **faux** sinon.

Le prédicat PERMUTE : SEG --> B

permute(<suite>)

Soit E, l'environnement et E_0, l'environnement initial.

Le prédicat PERMUTE appliqué au segment S_1 renvoie une valeur v de type booléen telle que v vaut **vrai** si les éléments du segment S_1 dans l'environnement E représentent une permutation des éléments du segment S_1 dans l'environnement E_0, et **faux** sinon.

Chapitre II. Définition du langage

Pour permettre de vérifier les assertions, il a fallu définir un langage de description des assertions et des instructions.

Un programme est composé de deux parties principales :

- la partie des déclarations,

- la partie des instructions.

Dans la première partie, on va trouver la déclaration des variables, des tableaux et des segments qui seront utilisés par les instructions ainsi que les expressions qui doivent être vérifiées (comme l'invariant, la précondition, la postcondition, ...).

Dans la deuxième partie, on trouvera les instructions du programme qui doivent permettre de trouver la véracité de la postcondition ainsi que de l'invariant à chaque exécution de la boucle.
Ce sont des instructions simples (affectation, test et branchement à une étiquette, ...).

Ces différentes instructions sont traduites dans une représentation interne plus facilement manipulable que des chaînes de caractères. Il y a donc une traduction du texte avant de pouvoir l'interpréter. Lors de la traduction d'une expression, par exemple, je dois pouvoir trouver tous les objets qui sont utilisés. J'ai donc imposé que tout objet soit déclaré avant d'être utilisé.

1. Le langage de déclaration des variables

On peut déclarer des variables, des tableaux, des segments ou des expressions. Il n'est pas nécessaire de grouper les déclarations par genre. On peut mélanger différents types de déclarations, mais tout objet utilisé par une déclaration DOIT ETRE déclaré avant d'être utilisé.
La zone des déclarations doit être terminée par le mot "*enddeclare*". Après ce mot, on ne peut plus mettre de déclaration.

Syntaxe

```
<partie déclaration> ::= <liste déclaration> enddeclare |
                         enddeclare
```

<liste déclaration> ::= <liste déclaration> < déclaration> ¦
<déclaration>

<déclaration> :: <déclaration variable entière> ¦
<déclaration variable logique> ¦
<déclaration tableau> ¦
<déclaration segment> ¦
<déclaration expression>

<déclaration variable entière> ::= declare <nom> as integer ¦
declare <nom> as integer <valeur entière>

Une variable entière doit posséder un nom "<nom>" et peut être initialisée à une valeur "<valeur entière>". Si aucune valeur n'est donnée, la variable prend une valeur par défaut qui n'appartient pas à l'ensemble des valeurs de la variable.

<déclaration variable logique> ::= declare <nom> as boolean ¦
declare <nom> as boolean <valeur de vérité>

Une variable booléenne doit posséder un nom "<nom>" et peut être initialisée à une valeur "<valeur de vérité>". Si aucune valeur n'est donnée, la variable prend une valeur par défaut, qui n'appartient pas à l'ensemble des valeurs de la variable.

<déclaration tableau> ::=
declare <nom> as array from <valeur entière>
to <valeur entière> ¦
declare <nom> as array from <valeur entière>
to <valeur entière> <liste valeurs entières>

Un tableau doit posséder un nom "<nom>" et peut contenir un nombre maximum de valeur. Ces valeurs sont accessibles via un numéro d'index. On doit donner la plus petite valeur de cet index et la plus grande. Le nombre d'éléments du tableau sera donc "max - min + 1".
Il est possible d'initialiser le tableau. Si le nombre de valeurs données ne correspond pas à la dimension du tableau, les dernières sont complétées par une valeur signifiant que cette partie du tableau n'a pas été initialisée. Si trop de valeurs sont fournies, les données excédentaires sont ignorées.

<déclaration segment> ::=
declare <nom> as segment from <nom tableau>
between <expr arith> & <expr arith>

Un segment est une partie d'un tableau. Il faut lui donner un nom, ainsi que le nom du tableau dans lequel il s'inscrit. Les bornes du segment sont des expressions dont les valeurs doivent se trouver entre les bornes du tableau dans lequel le segment est inscrit.

```
<liste valeurs entières> ::= <valeur entière> |
                <liste valeurs entières> <valeur entière>

<valeur entière> ::= -32500 .. +32500

<valeur de vérité> ::= vrai | faux

<déclaration expression> ::=
                declare <nom> as expression <expr arith> |
                declare <nom> as expression <expr logique>
```

Une expression est soit de type arithmétique, soit de type logique. Elle sera de type logique si elle contient un opérateur logique ou une fonction de type booléen. Sinon elle sera de type arithmétique.
La valeur de vérité '*vrai*' est associée à la valeur 1 tandis que '*faux*' est associée à 0.

Il est possible de définir des expressions qui sont des sous-expressions d'une autre plus complexe. Cette subdivision donnera un texte plus lisible

```
<expr arith> ::= <terme> |
                <expr arith> <operateur additif> <terme>

<terme> ::= <facteur> |
            <facteur> <opérateur multiplicatif> <terme>

<facteur> ::= <variable entière> |
              ( <expr arith> ) |
              -<facteur> |
              <constante> |
              <appel de fonction de type entier>

<opérateur additif> ::= + | -

<operateur multiplicatif> ::= * | div | mod

<operateur relationnel> ::= = | <= | >= | <> | < | >

<expression booléenne simple> ::=
              <variable booléenne> |
              <valeur de vérité> |
              <appel de fonction de type booléen> |
              ( <expr logique> )

<proposition atomique> ::=
              <expression booléenne simple> |
              <expr arith> <operateur relationnel> <expr arith>
```

```
<négation> ::= <proposition atomique> |
               not <proposition atomique>

<conjonction> ::= <négation> |
                  <négation> and <conjonction>

<expr logique> ::= <conjonction> |
                   <conjonction> or <expr logique> |
                   pourtout <variable entière> <expr arith> :
                    <expr arith> : <expr logique> |
                   ilexiste <variable entière> <expr arith> :
                        <expr arith> : <expr logique>
```

2. Instructions de programmation

Le jeu des instructions est limité à des instructions simples que l'on peut retrouver dans presque tous les langages de programmation : affectation, test, branchement, ...

Il y a également des instructions d'impression pour afficher le contenu des variables et pour évaluer les expressions.

Affectation

L'instruction d'affectation sert à donner une valeur à une variable simple. La valeur est le résultat de l'évaluation d'une expression.

Syntaxe :

```
<affectation> ::= let <variable simple> <expression>

<expression> ::= <expr arith> | <expr logique>

<variable simple> ::= <nom variable> |
                      <nom tableau> [ <expr arith> ] |
                      <nom segment> [ <expr arith> ]
```

Pour initialiser un tableau, il faut initialiser tous les éléments un par un grâce à une boucle. Il n'est pas possible d'initialiser un tableau ou segment grâce à un tableau ou segment de même taille.

Etiquette

Les étiquettes servent à placer des marques dans le programme pour permettre de réaliser des branchements conditionnels ou inconditionnels.

Syntaxe :

<étiquette> ::= etiq <valeur positive>

<valeur positive> ::= 0..32500

Cette instruction n'est pas traduite, mais marque seulement la place lors de la compilation du programme.

Branchement conditionnel

Il s'agit d'un changement du cours de l'exécution du programme vers une autre partie sous la contrainte d'une expression. Si cette expression est vrai, le programme se poursuit à l'étiquette qui est fournie, sinon, on continue à l'instruction suivante ce branchement conditionnel.

Syntaxe :

<condition> ::= if <expr logique> **goto** <etiq>

<etiq> doit être définie grâce à l'instruction <étiquette> ci-dessus.
Une référence à une étiquette qui n'existe pas ne peut être détecté que lors de l'exécution du programme, car on peut utiliser une étiquette avant de la définir (aller plus loin dans l'exécution du programme à la suite d'un test).

Branchement inconditionnel

Il s'agit de changer le cours de l'exécution du programme sans aucune contrainte. La prochaine instruction se trouve après l'étiquette qui est fournie.

Syntaxe :

<branchement> ::= goto <etiq>

Evaluation

Cette instruction permet d'évaluer une expression définie dans la partie déclaration.

Syntaxe :

<evaluation> ::= <nom expression>

<nom expression> est l'identifiant qui a été attribué à l'expression dans la première partie du fichier.

Affichage de données

Le langage dispose de trois instructions d'affichage :

- imprimer du texte
- imprimer une variable
- passer une ligne

a. Impression de texte

Il s'agit d'imprimer un texte défini par l'utilisateur. Le texte doit se trouver entre deux caractères identiques. Tout le texte compris entre ces deux caractères sera imprimé.

<print texte> ::= **print** <car> <texte> <car>

b. Impression de variable

Cette instruction permet d'afficher le contenu de variable simple. Elle ne permet pas d'afficher le contenu d'un tableau en entier.

<print variable> ::=
printvar <variable entière> |
printvar <variable logique> |
printvar <nom tableau> [<expr arith>]

Pour afficher le contenu d'un tableau, il faut créer une boucle avec deux étiquettes, un branchement conditionnel, un branchement inconditionnel et une affectation.

c. Passer une ligne

Cette instruction permet d'aérer la sortie des données. Son effet est de passer à la ligne suivante.

<passe ligne> ::= printnl

Lecture de données

Cette instruction de lecture permet d'initialiser des variables avant l'exécution du programme. On peut donc exécuter le programme sur différentes valeurs sans changer le fichier de départ du programme (zone des déclarations).

Syntaxe :

```
<read variable> ::=
                read <variable entière> |
                read <variable logique> |
                read <nom tableau> [ <expr arith> ]
```

Il n'est possible de lire que des nombres entiers. Pour affecter une valeur à une variable logique, il faut utiliser la convention définie plus haut : 1 pour vrai et 0 pour faux.

Vider le buffer d'entrée

L'instruction suivante permet de vider le buffer d'entrée.

<vider> ::= flush

Instruction d'arrêt

Cette instruction arrête l'exécution du programme que vous avez spécifié.

<arret> ::= stop

Quitter le programme

Cette instruction permet de quitter le programme d'évaluation et de retourner au système d'exploitation.

<quitter> ::= quit

Exécution du programme

La commande suivante permet de démarrer l'exécution du programme que vous avez spécifié. La première instruction est celle qui se trouve juste après la zone des déclarations des différents objets manipulés.

<execution> ::= run

Remarque

Si le premier caractère non blanc d'une ligne est le point-virgule, le reste de la ligne est ignoré. Ceci permet de placer des commentaires dans le code source. Ces commentaires peuvent se trouver dans la zone des déclarations ou dans le programme.

Programme

Un programme n'est qu'une suite des instructions définies ci-dessus.

<programme> ::= <suite instructions> **endprog** <run_quit>

<suite instructions> ::=
    <suite instructions> <instruction> |
    <instruction>

```
<instruction> ::= <affectation> |
                  <etiquette> |
                  <condition> |
                  <branchement> |
                  <evaluation> |
                  <print texte> |
                  <print variable> |
                  <passe ligne> |
                  <read variable> |
                  <vider> |
                  <arret>


<run_quit> ::= <run> <run_quit> | <quit>
```

Si le programme ne possède pas d'instruction '<arret>', il s'arrêtera lorsqu'il atteindra l'instruction **endprog**.

L'instruction '<execution>' ne donne pas lieu à une représentation interne, mais au démarrage immédiat de l'interprétation du code traduit, à partir de la première instruction. Si on place cette instruction dans le programme, ce doit être une des dernières instructions à traduire.

L'instruction '<quitter>', si elle se trouve dans le programme doit toujours être la dernière instruction.

Chapitre III. Exemples

Les exemples repris ci-dessous proviennent du mémoire de Jeannine Rulkin.

1. Programme de contraction d'un vecteur d'entiers triés.

1.1. Spécification

Soit a[1..n], n ≥ 1, un vecteur d'entiers triés par ordre croissant. Modifier le contenu d b[1..n] de sorte que b[1..j], où 1 ≤ j ≤ n, contienne les valeurs de a[1..n] triées par ordre croissant et sans répétitions.

Précondition

- n ≥ 1
- a trié par ordre croissant : a[1] ≤ a[2] ≥ ... a[n]

Postcondition

- 1 ≤ j ≤ n
- les j premiers éléments de b rangés par ordre strictement croissant :

  b[1] < b[2] < ... < b[j]

- {a[1..n]} = {b[1..j]}
- a inchangé

Invariant

- 1 ≤ j ≤ n
- les j premiers éléments de b sont rangés par ordre strictement croissant :

  b[1] < b[2] < ... < b[j]

- {a[1..i]} = b{[1..j]}

Instructions

Le programme Pascal permettant de réaliser le problème en fonction des spécifications est le suivant :

```
i := 1;
j := 1;
b[1] := a[1];

while( i <> n ) do
begin
     i := i + 1;

     if b[j] <> a[i] then
     begin
          j := j + 1;
          b[j] := a[i]
     end;
end;
```

1.2. Spécification dans le langage

Précondition

- tricrois(a)

- n ≥ 1

Postcondition

Soit

- b1, un segment de b entre 1 et j

- b2, un segment de b entre j+1 et n

On a :

- inchangé(a)

- pourtout i1 1 : n : ilexiste i2 1 : j : a[i1] = b[i2]

- tristrcrois(b1)

- pourtout i1 1 : n : ilexiste i2 1 : n : b[i1] = a[i2]

- j >= 0

- j <= n

Situation générale

Soit

- a1, segment de a entre 1 et i
- a2, segment de a entre i+1 et n

on a :

- pourtout i1 1 : i : ilexiste i2 1 : j : a[i1] = b[i2]
- i >= 1
- j <= n
- i >= j
- tristrcrois(b1)
- pourtout i1 1 : j : ilexiste i2 1 : i : b[i1] = a[i2]
- j >= 1
- j <= n

1.3. Encodage

```
;
; copier tous les éléments d'un tableau sans répétition
;

declare a as array from 1 to 8   1 2 2 3 3 3 4 5
declare b as array from 1 to 8

declare n as integer 8
declare i as integer
declare j as integer

declare b1 as segment from b between 1 & j

declare precondition as expression (n >= 1) and tricrois( a )

declare i2 as integer
declare i1 as integer

declare p1 as expression inchange( a )
declare p2 as expression pourtout i1 1 : n : ilexiste i2 1 : j
          : a[i1] = b[i2]
declare p3 as expression tristrcrois( b1 )
declare p4 as expression pourtout i1 1 : j : ilexiste i2 1 : n
          : b[i1] = a[i2]
declare p5 as expression (i >= 0) and (j <= n)
```

```
declare postcondition as expression p1 and p2 and p3 and p4
          and p5


declare inv1 as expression pourtout i1  1 : i : ilexiste i2 1
          : j : a[i1] = b[i2]
declare inv2 as expression (1 <= i) and (i <= n) and (i >= j)
declare inv3 as expression tristrcrois( b1 )
declare inv4 as expression pourtout i1 1 : j : ilexiste i2 1:
          i : b[i1] = a[i2]
declare inv5 as expression (1 <= j) and (j <= n)

declare inv as expression inv1 and inv2 and inv3 and inv4 and
          inv5

declare B as expression i = n

enddeclare

print "n ? "
read n
flush
printnl
printnl
eval n

eval precondition
let i 1
let j 1
let b[1] a[1]

etiq 1
eval B
eval inv

if i = n goto 999

let i i+1

if b[j] = a[i] goto 1

let j j+1
let b[j] a[i]

goto 1

etiq 999

let i1 1
printnl
print "Résultat"
printnl

etiq 9991

if i1 > j goto 9992
printvar b[i1]
print " "
let i1 i1+1
```

```
goto 9991

etiq 9992
printnl
eval j
eval B
eval inv
eval postcondition

stop

run

quit
```

2. Programme permutant un vecteur, en plaçant ses éléments strictement négatifs à gauche et les autres, à droite

2.1. Spécification

Soit a[1..n], un tableau d'entiers initialisés. Permuter ce dernier de sorte que, à gauche se trouvent ses éléments négatifs et à droite ses éléments positifs ou nuls.

Précondition

- a[1..n] initialisé

Postcondition

- a[1..n] permuté de sorte que :
  il existe i_S : $1 \leq i_S \leq n$:
  (pour tout k : $1 \leq k \leq i_S$: a[k] < 0
  et
  pour tout q ; $i_S+1 \leq q \leq n$: a[q] $\geq$ 0)

Invariant

- $1 \leq i \leq j+1 \leq n+1$
- a[1..n] permuté de sorte que
  - pour tout k : $1 \leq k \leq i-1$: a[k] < 0
  - pour tout q : $j+1 \leq q \leq n$: a[q] $\geq$ 0

Instructions

```
i := 1;
j := n;

while( i <= j )
begin
     if a[i] < 0 then i := i + 1
     else if a[j] >= 0 then j := j - 1
     else
     begin
          v := a[i];
          a[i] := a[j];
          a[j] := v;
          i := i+1;
          j := j-1;
     end;
end;
```

2.2. Spécification dans le langage.

Précondition

Postcondition

- permute(a)
- pourtout k 1 : i : a[k] < 0
- pourtout q i+1 : n : a[q] >= 0

Situation générale

- permute(a)
- pourtout k 1 : i-1 : a[k] < 0
- i >= 1
- i <= j+1
- j <= n
- pourtout q j+1 : n : a[q] >= 0

2.3. Encodage

```
;
; illustration 6 : page 37
;
declare a as array from 1 to 10   9 -2 8 7 -5 6 -7 8 10 -11

declare i as integer
declare j as integer

declare n as integer 10

declare B as expression i > j

declare i1 as expression permute( a )

declare k as integer
declare q as integer

declare i2 as expression pourtout k  1 : i-1 : a[k] < 0

declare i3 as expression (i >= 1) and (i <= j+1) and (j <= n)

declare i4 as expression pourtout q  j+1 : n : a[q] >= 0
```

```
declare inv as expression i1 and i2 and i3 and i4

declare post as expression i1 and i2 and i4 and B

;
; variable temporaire
;

declare v as integer

enddeclare

let i 1
let j n

etiq 1

eval B
eval inv

if i > j goto 999

if a[i] < 0 goto 3

if a[j] >= 0 goto 4

;
; il faut permuter a[i] et a[j]
;
let v a[i]
let a[i] a[j]
let a[j] v
let i i+1
let j j-1
goto 1

etiq 3
let i i+1
goto 1

etiq 4
let j j-1
goto 1

etiq 999

printnl
printnl
print "Résultat"
printnl
print "a = "

let k 1

etiq 9991
if k > n goto 9992

printvar a[k]
print " "
let k k+1
```

```
goto 9991

etiq 9992
printnl
eval B
eval inv
eval post
eval i1
eval i2
eval i3
eval i4

stop

run

quit
```

3. Programme de recherche dans un tableau du segment de somme maximale.

3.1. Spécification.

Précondition

- a[1..n] initialisé

Postcondition

- a inchangé
- $1 \leq d \leq f+1 \leq n+1$
- pour tout i,j : $1 \leq i \leq j+1 \leq n+1 : \sum_{k=i}^{j} a[k] \leq \sum_{k=d}^{f} a[k]$

Invariant

- $1 \leq k \leq i+1 \leq n+1$
- $1 \leq d \leq f+1 \leq i+1$
- $\sum_{p=d}^{f} a[p] = m$
- $\sum_{p=k}^{i} a[p] = s$
- pour tout d', f' : $1 \leq d' \leq f'+1 \leq n+1 : \sum_{p=d'}^{f} a[k] \leq m$
- pour tout k' : $1 \leq k' \leq i+1 : \sum_{p=k'}^{i} a[k] \leq s$

3.2. Spécification dans le langage.

Précondition

Postcondition

- inchangé(a{1:n})
- somme(a{d:f}) = m

- d >= 0
- d <= n

- f >= 0
- f <= n

- d <= f

- pourtout k 1 : n : pourtout l k : n : somme(a{k:l}) <= m

Situation générale

- somme(a{d:f}) = m

- d >= 1
- d <= f+1
- f <= i+1

- i >= 1
- i <= n

- pourtout k 1 : i : pourtout l k : i : somme(a{k:l}) <= m

- i >= 1
- i <= n

- somme(a{k:i}) = s
- k >= 1
- k <= n+1
- i <= n

- pourtout k 1 : i : pourtout l k : i : somme(a{k:l}) <= s

Programme

```
d := 1;
f := 0;
m := 0;
k := 0;
i := 0;
s := 0;

while i<>n do
     begin
          i := i + 1;
          s := s + a[i];

          if s <= 0 then
               begin
                    k := k + 1;
                    s := 0;
               end;

          if m < s then
               begin
```

```
                    m := s;
                    d := k:
                    f := i;
                end;
    end;
```

3.3. Encodage.

```
declare a as array from 1 to 10   5 96 -8 100 5 9 74 -100 9 10

declare d as integer
declare f as integer
declare i as integer
declare s as integer
declare m as integer
declare k as integer

declare size as integer 10

declare i1 as expression (1 <= k) and (k <= i+1) and (i+1 <=
          size+1)
declare i2 as expression (1 <= d) and (d <= f+1) and (f+1 <=
          size+1)
declare i3 as expression somme( a{d:f} ) = m
declare i4 as expression somme( a{k:i} ) = s

declare d1 as integer
declare f1 as integer
declare k1 as integer

declare i5 as expression pourtout d1 1 : i+1 : pourtout f1
          d1-1: i : somme( a{d1:f1} ) <= m
declare i6 as expression pourtout k1 1 : i+1 :
          somme( a{k1:i} ) <= s

declare inv as expression i1 and i2 and i3 and i4 and i5
          and i6

declare B as expression i = size

enddeclare

let size 10


let d 1
let f 0
let m 0
let k 1
let i 0
let s 0

etiq 1

if i = size goto 99
```

```
printnl
eval B
eval inv

let i i+1
let s s+a[i]

if s >= 0 goto 2
let k i+1
let s 0
etiq 2

if m >= s goto 3
let d k
let f i
let m s
etiq 3

goto 1

etiq 99

printnl
print "Terminaison"
printnl
printnl
eval inv
print "m = "
printvar m
printnl
print "d = "
printvar d
print "  --  f = "
printvar f
printnl

stop

endprog

run

quit
```

Chapitre IV. Implémentation.

Dans cette partie, je vais expliquer la manière dont j'ai utilisé les différents concepts préconisés dans le langage de description défini par Jeannine Rulkin.

J'ai utilisé le langage C++ pour implémenter ces concepts. Il m'a semblé que ce langage pouvait fournir différentes facilités de programmation.

Le langage orienté objet définit trois techniques de programmation qui permettent d'organiser des types abstraits de données en relation les uns avec les autres, et d'exploiter leurs caractéristiques communes afin de réduire l'effort de programmation. Celle-ci se caractérise par :

- L'encapsulation des données qui permettent aux programmes clients de n'interagir avec des instances de classe que par le biais d'un ensemble bien défini d'opérations. Celles-ci isole les informations qu'elle contiennent et les algorithmes qu'elles utilisent.

- L'héritage, qui simplifie la création d'une nouvelle classe similaire à une classe existante en donnant au programmeur la possibilité de n'exprimer que les *différences* entre la nouvelle classe et la classe existante. De cette manière, il n'a pas à réécrire complètement la nouvelle classe.

- La liaison dynamique, parfois appelée *liaison retardée*, permet à chaque classe d'un groupe d'avoir sa propre implémentation d'une fonction particulière. Au moment de l'exécution, C++ détermine la classe spécifique de l'instance et appelle l'implémentation de la fonction dans cette classe.

J'utilise le plus souvent l'héritage et la liaison dynamique dans la réalisation de ce mémoire : on trouve plusieurs éléments ayant des caractéristiques communes, mais qui diffèrent sur la manière dont on accède à l'une ou l'autre (variable retournant une valeur entière ou logique, un tableau ou un segment, ...)

Il m'a semblé que ces deux caractéristiques principales du C++ permettront de simplifier la programmation des différents concepts définis dans le langages. Un tableau et un segment possèdent tous les deux des caractéristiques communes, bien que le deuxième soit une partie du premier, on accède de la même manière à un élément : grâce à son indice. Une variable entière va différer d'une variable logique par le type de valeur qui est retournée : un entier ou vrai/faux.

Une fonction commune a toutes les classes est 'eval'. Elle permet d'évaluer une instance de la classe. Pour une variable, c'est connaître la valeur qui lui est associée, pour une fonction, c'est le résultat de la fonction d'après les arguments qu'elle a reçu, ... Grâce à l'édition dynamique,

l'évaluation d'une expression ira chercher les bonnes fonctions nécessaire à l'évaluation des différents arguments de l'expression. Cette fonction est déclarée dans la classe de base, classe à partir de laquelle on dérive toutes les classes qui sont utilisées. Ainsi, toute occurrence d'une classe pourra faire appel à cette fonction, car elle doit être redéfinie dans la classe héritée, sinon elle utilise la valeur par défaut fournie par sa classe de base.

1. Introduction

Les expressions introduites par l'utilisateur sont fournies dans une représentation infixée. Par facilité pour l'évaluation de ces expressions, elles sont traduites sous le format postfixé. Cette représentation supprime tous les niveaux de parenthèse que l'utilisateur peut introduire pour changer l'ordre de calcul. Pour ces calculs, une pile est nécessaire pour stocker les différentes valeurs temporaires.

Le type des valeurs est l'entier (de -32500 à +32500), mais l'utilisateur pourrait avoir un résultat temporaire qui ne soit pas représentable sous format entier. Pour cette raison, la pile de valeur possède des arguments représentés sous le format d'entiers longs (32 bits au lieu de 16). Un calcul tel que "32000 * 32000 div 32000" donnera le bon résultat (soit 32000), au lieu d'une autre valeur quelconque.

La pile est représentée par un tableau d'entiers longs signés. Le nombre maximum d'éléments que l'on peut disposer sur la pile est de 100. Le nom du tableau est **PILE_VAL** et **SOMMET_PILE_VAL** indique la place où l'on peut placer le prochain élément. La constante **MAXPILEVAL** indique le nombre maximum d'éléments que peut contenir la pile.

```
#define MAXPILEVAL      100

long pile_val[ MAXPILEVAL ];

int sommet_pile_val = 0;
```

Tous les noms utilisés pour identificateur ont une longueur maximale de **MAXVARLEN**. Cette valeur est de 32 caractères.

Les valeurs des différentes données, tableaux et variables, sont conservées dans un tableau qui joue le rôle de la mémoire du programme. Il s'agit d'un tableau en C, 'mem', possédant un maximum de **MAXMEM** valeurs.

```
#define MAXMEM   1000

int mem[ MAXMEM ];

int nadr;
```

La variable '**nadr**' indique la prochaine place libre dans le tableau de la mémoire.

Il y a différentes fonctions qui vont utiliser ce tableau :

void init_mem()

initialise la variable **nadr**,

term_mem()

exécute les instructions de terminaison lorsque l'on quitte le programme,

int lire_mem(int adr)

permet d'obtenir la valeur se trouvant à l'adresse **adr** dans la mémoire

void ecrire_mem(int adr, int val)

permet d'écrire la valeur **val** à l'adresse **adr** de la mémoire.

Certaines fonctions ont besoin de connaître l'environnement initial du programme, c'est à dire l'état des variables et tableaux avant l'exécution du programme. Cet environnement initial sera un tableau, mais représenter sous la forme d'un pointeur : en effet, alors que l'on ne peut pas connaître le nombre de variables et les tailles des tableaux que l'utilisateur va définir, la taille de cet environnement est connu lorsque l'on commence l'exécution du programme (c'est indiqué par la variable **nadr**). Il suffit donc de réserver la place nécessaire pour définir le tableau de l'environnement initial. Celui-ci s'appelle **mem0**.

int *mem0;

L'accès aux données contenues dans ce tableau se fait via des fonctions C :

void init_mem0( )

initialise le tableau de l'environnement initial en fonction de l'environnement **mem** et de **nadr**

void term_mem0( )

Cette fonction va détruire le pointeur **mem0** crée par la fonction **init_mem**.

int lire_mem0(int adr)

Cette fonction va lire la valeur contenue à l'adresse **adr**.

Il n'est pas nécessaire de définir une fonction pour écrire dans ce tableau, car les valeurs sont celles de départ et ne doivent pas changer.

Il faut utiliser la procédure **init_mem0** avant de commencer l'exécution du programme.

Comme pour l'environnement initial, un tableau est utilisé pour vérifier si une variable a été examinée. Ce tableau est utilisé par la fonction **examine** qui porte sur les tableaux et segments. Chaque fois que l'on a accès à une variable ou à un élément d'un tableau, l'adresse correspondante dans le tableau **exa** est mise à jour pour indiquer que cet élément a été utilisé. Cette mise à jour est faite uniquement lors d'une lecture de la valeur de la variable. Le tableau est initialisé en même temps que l'environnement initial.

2. La classe de base.

Cette classe n'a aucune instance propre dans le programme. Elle ne sert que de classe de base pour les concepts du langage. Elle contient les méthodes que toute classe dérivée doit posséder.

```
class ClasseBase {
protected:
     int  err;

public:
               ClasseBase( ) {};
               ~ClasseBase( ) {};

virtual int          isA( ) const = 0;
virtual int          isEqual( const ClasseBase& ) const = 0;

ClasseBase&     operator=( const ClasseBase& ) {return *this;};

virtual    const int       eval( ListeLineaire *l = NULL ) = 0;
int        resultat( )
               { return (int)pile_val[ --sommet_pile_val ]; };

int        error( ) const { return err; };
void       error( int v ) { err = v; };

virtual    char       *nom( ) { return ""; };
           void       nom( char *) { };

};
```

La méthode 'isA' permet d'identifier le type de classe utilisée. Ceci est utile quand on a un pointeur vers cette classe de base. Le C++ va utiliser la méthode 'isA' qui correspond à l'instance que l'on utilise.

La méthode 'isEqual' permet de vérifier que deux instances d'une classe sont égales. En fonction de la classe, celles-ci seront identiques si elles sont de même type, ou bien si elles ont le même nom et le même type, ...

L'opérateur '=' permet de réaliser des affectations de valeurs dans la classe

La méthode 'eval' permet d'évaluer les différentes classes. Suivant le type de cette classe, l'évaluation va être différente. En effet, on évalue pas de la même manière une variable, une constante ou une fonction. Si l'évaluation est possible, le résultat de l'évaluation est placé sur la pile et la méthode retourne la valeur 0, sinon elle retourne la constante **ERROR**.

Cette méthode possède un argument qui est une liste linéaire pour permettre de passer les arguments qui sont nécessaires pour l'évaluation des fonctions : en effet, celles-ci ont besoin de 1 à 3 arguments dépendant du type de la fonction. Ces arguments sont placés dans une liste linéaire.

La méthode 'résultat' permet de récupérer le résultat de l'évaluation de l'instance de la classe.

Les instances de certaines classes possèdent un nom, comme les variables, les tableaux, les segments. La méthode 'nom' permet de nommer ces instances ou de connaître le nom d'une instance.

Si la méthode est utilisée sans argument, le nom associé à l'instance est retourné. Si un argument, qui est une chaîne de caractères, est donné, on affecte ce nom à l'instance.

3. Liste linéaire

Une liste linéaire est une collection d'objets. On accède séquentiellement à tous les objets de cette liste. A partir d'un élément, on ne peut passer que vers le suivant. Il est possible de se replacer en début de liste.

C'est sous forme de liste linéaire que sont conservés les différentes données de l'environnement. Si on a beaucoup de données, la recherche est séquentielle et peut prendre du temps, mais, comme cette recherche sera effectuée principalement par le compilateur, ce ne sera que temporaire.
Les expressions, que l'on définira plus loin, utilisent aussi ce concept de liste linéaire.

```
typedef struct typeartlistelin {
     struct typeartlistelin *suiv;
     ClasseBase *ref;
} TYPEARTLISTELIN;


class ListeLineaire : public ClasseBase {
protected:
     long n;
     struct typeartlistelin *e, *c;
public:
     ListeLineaire ( );
     ~ListeLineaire ( );


virtual int    isA( ) const { return _ClasseListeLin; };
virtual int    isEqual( const ClasseBase& ) const;

virtual   const int eval( ListeLineaire * = NULL )
                    { return ERROR; };

     int eof( );
     void creer( );
     int avancer( );
     int inserer( ClasseBase *V );
     ClasseBase *courant( );
     long nbr( ) const { return n; };
     void repositionner( );
     int supprimer( int all = 0 );
     void detruire( int all = 0 );

     ListeLineaire& operator=( ListeLineaire& L );
};
```

Le concept de liste linéaire est implémenté sous forme d'une liste chaînée. L'information qui est conservée est stockée sous forme de pointeur vers la donnée.

Pour réaliser la liste linéaire, je définis le type 'TYPEARTLISTELIN' qui contiendra l'information que l'on conserve. Cette donnée est conservée via son pointeur. Celui-ci est du type pointeur vers la classe de base. Il est donc

possible de stocker des instances de différentes classes dans la même liste linéaire. Ceci sera utilisé plus loin.

On utilise deux pointeurs pour retrouver l'information : le premier, 'e', indique le premier élément de la liste linéaire, le second, 'c', indique l'élément précédent le courant. Si la liste ne contient aucun article, 'c' doit se trouver sur l'élément précédent. Il faut donc créer un article qui ne contient aucune information, mais qui assure la cohérence de la liste. Cet article vide permet de gérer plus facilement l'insertion de données.

Méthodes :

ListeLineaire

Cette méthode est le constructeur qui est appelé automatiquement par le programme, lorsque celui-ci aperçoit une variable liste linéaire pour la première fois.
Cette méthode appelle automatiquement la méthode 'creer' définie ci-dessous.

~ListeLineaire

Cette méthode est le destructeur. Elle appelle la méthode 'detruire' définie ci-dessous et détruit le premier élément créé pour la gestion de la liste.

isA

Retourne la valeur qui identifie une liste linéaire.

isEqual

Retourne 1 si la liste est identique à la liste donnée en argument, 0 sinon. Deux listes sont identiques si tous les éléments d'une liste se retrouvent dans l'autre et dans le même ordre.
Pour vérifier que deux éléments sont identiques, on utilisera la méthode 'isEqual' de chaque classe correspondante.

eval

Cela n'a pas de sens d'évaluer une liste linéaire, car il s'agit d'une collection d'objets, sans aucun point commun apparent.
La méthode va donc retourner la constante **ERROR**.

eof

Cette méthode vérifie si on se trouve à la fin de la liste linéaire. Si oui, retourne 1, sinon retourne 0.

creer

Cette méthode va créer une liste linéaire. C'est cette méthode qui est appelée par le constructeur. Cette méthode initialise les différentes variables.

avancer

Cette méthode va changer l'élément courant de la liste linéaire. Elle passe à l'élément suivant directement le courant. La méthode suppose que l'on ne se trouve pas en fin de liste.
Retourne 1 si on a pu changer le pointeur vers l'élément suivant, 0 sinon.

inserer

Cette méthode va insérer l'élément à conserver avant l'élément courant. Si on se trouvait en fin de liste, il est ajouté à la fin de celle-ci.
Comme on utilise une liste chaînée, il faut trouver l'élément qui est avant le courant pour mettre à jour le pointeur de liste. C'est pour cela que l'on conserve le pointeur vers l'élément précédent plutôt que vers le courant.

courant

Cette méthode permet d'obtenir l'adresse de l'information qui est contenue dans l'élément courant de la liste. C'est un pointeur vers la classe de base

nbr

Retourne le nombre d'éléments qui se trouvent dans la liste linéaire.

repositionner

Cette méthode positionne le pointeur vers l'élément courant en début de liste.

supprimer

Cette méthode supprime l'élément courant de la liste. Seul l'élément de la liste linéaire est détruit, mais pas l'information qui est conservée. En effet, dans l'élément de la liste, on a un pointeur vers l'information. Comme une même

donnée peut se trouver dans deux listes différentes, on n'y place qu'un pointeur vers la donnée.

Si on donne un argument non nul, la donnée pointée par l'élément courant est supprimée également.

detruire

Cette méthode va supprimer tous les éléments de la liste linéaire. Toutes les données sont supprimées, sauf le premier élément de la liste. Il n'est donc pas nécessaire de rappeler la méthode 'creer' avant de réutiliser la liste linéaire.

Si on donne un argument non nul, la donnée pointée par l'élément courant est supprimée également.

operator =

Cet opérateur permet de copier tous les éléments d'une liste dans une autre. Après cette copie, on aura deux listes dont les éléments sont des pointeurs vers une même donnée.

4. Les fonctions prédéfinies.

Cette classe dérivée de la classe de base permet d'évaluer les différentes fonctions et opérateurs qui sont définies dans le langage3.

Tous les opérateurs et fonctions sont regroupés sous une seule et même classe qui possède un argument indiquant de quel fonction il s'agit.

```
class ClasseFonction : public ClasseBase {
protected:

    int         genre; // Numéro de la fonction

public:
                ClasseFonction( int n = 0 ) { genre = n; };
                ~ClasseFonction( ) { genre = 0; };

virtual   int   isA( ) const { return _ClasseFonction; };
virtual   int   isEqual( const ClasseBase& ) const;

const     int   eval ( ListeLineaire * = NULL );
};
```

La classe fonction est également dérivée de la classe de base. On va donc retrouver les deux méthodes 'isA' et 'isEqual' qui permettent d'identifier une instance de cette classe.

La méthode 'eval' aura un comportement qui dépend du paramètre **genre** définissant le type de fonction représenté par l'instance de cette classe. Ce paramètre est mis-à-jour via le constructeur **ClasseFonction**. Une constante est associée à chaque opérateur ou fonction et c'est cette valeur qui sera placée dans l'instance de la classe. La méthode 'eval' est donc composée d'un 'case' sur le genre (opérateur ou fonction) et sur le nombre d'arguments intervenant dans l'évaluation.

La liste linéaire fournie en paramètre contient les arguments nécessaires pour la bonne évaluation des fonctions. Il s'agit de passer des tableaux ou des segments pour ces fonctions. On donne donc les adresses des arguments utilisés et dans l'ordre défini par les fonctions : 'segment(S1, S2)' vérifier si le segment S2 est inclus dans le segment S1.

Les opérateurs vont chercher leurs arguments sur la pile des valeurs et placent le résultat sur cette même pile. Les fonctions utilisent la liste linéaire pour trouver les arguments et placent le résultat sur la pile. Il y deux fonctions qui ne respectent pas cette règle : il s'agit des fonctions **'head'** et **'tail'** qui ne retournent pas une valeur entière ou logique, mais un segment. La première retourne un segment moins son dernier élément, tandis que la deuxième retourne le segment moins le premier élément. Pour réaliser cela, lorsque le compilateur repère une de ces fonctions dans une expression, il génère un segment qui sera utilisé pour récupérer le résultat. Ces deux fonctions ont un argument

supplémentaire : le segment dans lequel il faut placer le résultat.

La fonction **examine** utilise le tableau **exa**. La fonction **permute** utilise l'environnement initial.

Il existe deux opérateurs autres que les opérateurs standards définis dans le premier chapitre : le "pour tout" et "il existe". Le premier permet d'exprimer une condition sur tous les éléments d'un segment ou d'un tableau, tandis que le second permet d'exprimer qu'un des éléments d'un segment ou d'un tableau doit vérifier une condition. Ces opérateurs ne sont présents que dans les expressions et ont la syntaxe suivante :

<pour tout> ::= pourtout <variable entière> <expression 1> :
<expression 2> : <expression3>

<il existe> ::= ilexiste <variable entière> <expression 1> :
<expression 2> : <expression3>

où <variable entière> est une variable qui servira à exprimer l'indice de l'élément du tableau ou du segment qui doit vérifier l'expression <expression 3>, cet indice qui doit se trouver entre les bornes <expression 1> et <expression 2>. Les expressions des bornes sont du genre entier, tandis que l'expression de la condition est du genre logique.

Pour l'évaluation de 'pourtout', tous les éléments sont vérifiés, à moins que l'on ait trouvé un qui ne vérifie pas la condition. Pour l'évaluation de 'ilexiste', on s'arrête dès qu'un élément vérifie l'expression <expression 3>.

Pour le compilateur, tout ce qu'il trouve après le ':' de l'expression <expression 2> va faire partie de la condition. Si l'on a une expression du style :

```
ilexiste i 1 : n : ( ( pourtout k 1 : i : a[k] < 0 ) and
                     ( pourtout q i+1 : n : a[q] >= 0 ) )
```

il est préférable de décomposer la condition de l'opérateur 'ilexiste' en deux expressions et de donner l'expression suivante pour 'ilexiste' :

```
expr i1 = pourtout k 1 : i : a[k] < 0

expr i2 = pourtout q i+1 : n : a[q] >= 0

ilexiste i 1 : n : i1 and i2
```

Cette deuxième formulation est plus claire et l'utilisateur risque moins de faire des erreurs ou de se poser de questions sur la portée de tel ou tel opérateur 'pourtout' ou 'ilexiste'. En découpant, il sera sur de cette portée.

Ci-dessous se trouvent les classes des opérateurs 'pourtout' et 'ilexiste'. Elles sont dérivées de la classe de base 'ClasseBase'.

```
class ClassePourTout : public ClasseBase {
protected:
    ClasseVarEntiere   *v;
    ClasseExpression    bi, // Valeur inférieure de la boucle
                        bs, // Valeur supérieure de la boucle
                        exp;

public:
                ClassePourTout( ClasseVarEntiere *ve = NULL,
                          ClasseExpression *bil = NULL,
                          ClasseExpression *bsl = NULL,
                          ClasseExpression *expl = NULL )
                     {
                       if( ve ) v = ve;
                       if( bil ) bi = *bil;
                       if( bsl ) bs = *bsl;
                       if( expl ) exp = *expl;
                     };
                ~ClassePourTout( );

virtual   int  isA( ) const { return _ClassePourTout; };
virtual   int  isEqual( const ClasseBase& ) const;

const     int  eval ( ListeLineaire * = NULL );

virtual   char      *nom( ) { return "Pour tout"; };

};


class ClasseIlExiste : public ClasseBase {
protected:
    ClasseVarEntiere *v;
    ClasseExpression  bi, // Valeur inférieure de la boucle
                      bs, // Valeur supérieure de la boucle
                      exp;

public:
                ClasseIlExiste( ClasseVarEntiere *ve = NULL,
                          ClasseExpression *bil = NULL,
                          ClasseExpression *bsl = NULL,
                          ClasseExpression *expl = NULL )
                     {
                       if( ve ) v = ve;
                       if( bil ) bi = *bil;
                       if( bsl ) bs = *bsl;
                       if( expl ) exp = *expl;
                     };
                ~ClasseIlExiste( );

virtual   int  isA( ) const { return _ClasseIlExiste; };
virtual   int  isEqual( const ClasseBase& ) const;

const     int  eval ( ListeLineaire * = NULL );

virtual   char      *nom( ) { return "Il existe"; };
};
```

Les constructeurs des deux classes initialisent les différents variables : variable à utiliser, expression de la borne inférieure et de la borne supérieure, condition à vérifier.

La méthode **isA** retourne l'identifiant de la classe.

La méthode **isEqual** retourne 1 si l'argument utilise la même variable de boucle, possède les mêmes expressions de bornes et la même expression de condition.

5. Les variables

Le langage définit deux genres de variables : les variables entières et variables logiques. Toutes les deux possèdent un nom et une adresse dans la mémoire du programme.
L'adresse de la variable n'a pas une place bien précise : on mélange les tableaux et les variables dans la mémoire.

Il y a deux classes définies pour ces variables, toutes dérivées d'une classe commune définissant les variables.

Classe ClasseVariable

Dans cette classe, on retrouve les méthodes communes aux deux types de variables.
On retrouve le nom et l'adresse de la variable en mémoire.

```
class ClasseVariable : public ClasseBase {
protected:
     char n[MAXVARLEN + 1];
     int adr;

public:
               ClasseVariable( const char *p = NULL );
               ~ClasseVariable( );

virtual    int  isA( ) const { return _ClasseVariable; };
virtual    int  isEqual( const ClasseBase& ) const;

const      int       eval ( ListeLineaire * = NULL );

virtual    char      *nom( ) { return n; };
           void      nom( char *p ) { strcpy( n, p ); };

           void      adresse( int _adr );
           int       adresse( void ) const;
};
```

Méthodes :

isA

Identifie une instance de cette classe. Il ne doit jamais avoir d'instance de cette classe, car cette variable ne possède aucun type.

isEqual

Cette méthode vérifie si on a bien la bonne variable, c'est-à-dire si le nom de l'argument est identique au nom de l'instance. Cette méthode sera utilisée par les instances des classes dérivées.

eval

Retourne ERROR, car cette classe ne possédant pas de type, il ne peut y avoir d'évaluation.

char *nom

Fournit le nom de la variable dans l'environnement courant.

void nom(char *p)

Nomme la variable avec le nom pointé par p.

void adresse(int _adr)

Associe la variable à l'adresse **_adr** de la mémoire du programme.

int adresse( )

retourne l'adresse associée à la variable.

Classe ClasseVarEntiere

Cette classe va définir ce qui concerne les variables entières. On va trouver les opérateurs d'affectation et d'évaluation de ces variables.
Cette classe est dérivée de la classe **ClasseVariable**.

```
class ClasseVarEntiere : public ClasseVariable {
protected:

public:
                ClasseVarEntiere( const char *p = NULL ) :
                          ClasseVariable( p )     {};
                ~ClasseVarEntiere( ){};

virtual    int  isA( ) const { return _ClasseVarEntiere; };

           int  operator=( const ClasseVarEntiere& t );
           int  operator=( int i );

const      int        eval ( ListeLineaire * = NULL );
};
```

Méthodes :

isA

retourne l'identifiant de la classe.

ClasseVarEntiere

Constructeur associant le nom à la variable.

int operator=(const ClasseVarEntiere& t)

Opérateur qui permet d'affecter la variable entière t à l'instance de la variable courante. Cet opérateur retourne la valeur de la variable.

int operator=(int i)

Opérateur permettant d'affecter la valeur de la variable i à la variable entière représentée par la classe. Cet opérateur retourne la valeur de i.

eval

Effectue l'évaluation de la variable, c'est-à-dire place la valeur associée à la variable sur la pile des valeurs.

Classe ClasseVarLogique

Cette classe va définir ce qui concerne les variables logiques. On va trouver les opérateurs d'affectation et d'évaluation de ces variables.
Cette classe est dérivée de la classe **ClasseVariable**.

La valeur 1 est associée à **vrai**, et 0 pour **faux**.

```
class ClasseVarLogique : public ClasseVariable {
protected:

public:
                ClasseVarLogique( const char *p = NULL ) :
                          ClasseVariable( p ) { };
                ~ClasseVarLogique( );

virtual    int  isA( ) const { return _ClasseVarLogique; };

           int  operator=( const ClasseVarLogique& t );
           int  operator=( int i );
```

```
const     int  eval ( ListeLineaire * = NULL );
};
```

Méthodes :

isA

retourne l'identifiant de la classe.

ClasseVarLogique

Constructeur associant le nom à la variable.

int operator=(const ClasseVarLogique& t)

Opérateur qui permet d'affecter la valeur de la variable logique t à l'instance de la variable logique courante. Cet opérateur retourne la valeur de la variable.

int operator=(int i)

Opérateur permettant d'affecter la valeur de la variable i à la variable entière représentée par la classe. Cet opérateur retourne la valeur de i.

eval

Effectue l'évaluation de la variable, c'est-à-dire place la valeur associée à la variable sur la pile des valeurs.

6. Les constantes

Cette classe est utilisée pour stocker les nombres que l'on rencontre dans les expressions. Ces nombres ne peuvent pas être placés dans la zone des variables, car il y a un nombre limité d'éléments. On utilise donc une variable dynamique en C vers un entier pour enregistrer la valeur de la constante.

La méthode **eval** définie dans cette classe va placer la valeur de l'expression sur la pile des valeurs. Dans le cas d'une valeur logique, la convention ci-dessus est utilisée.

La valeur de la constante est donnée dans le constructeur **ClasseConstante**.

```
class ClasseConstante : public ClasseBase {
protected:
    int v;

public:
                ClasseConstante( int val = 0 ) { v = val; };
                ~ClasseConstante( ) {};

virtual   int   isA( ) const { return _ClasseConstante; };
virtual   int   isEqual( const ClasseBase& ) const;

const     int         eval ( ListeLineaire * = NULL );

          int         operator=( int val );
          int         operator=( const ClasseConstante& c );

};
```

7. Les tableaux et segments

Tableaux et segments possèdent les mêmes caractéristiques mais les seconds font parties des premiers. Les fonctions d'accès à un élément sont identiques, chacun possède des bornes, ... Il est donc intéressant de définir une classe "**Tableau-Segment**" qui regroupe les méthodes utilisées par les tableaux et les segments. C'est ce qui est fait avec la classe **ClasseTabSeg.**

Les classes **ClasseTableau** et **ClasseSegment** reprendront les fonctions caractéristiques de tableaux et des segments.

La plupart des expressions utilisent un élément du tableau ou d'un segment. Pour avoir accès à un tel élément, il y a la classe **ClasseVarInd** qui permet d'accéder à un élément.

Classe ClasseTabSeg

```
class ClasseTabSeg : public ClasseBase {
protected:
     char n[MAXVARLEN + 1];
public:
                    ClasseTabSeg( const char *q = NULL )
                    { if( q ) strcpy( n, q );
                      else strcpy( n, "" );
                    };
                    ~ClasseTabSeg( ) {};

virtual    int      isA( ) const { return _ClasseTabSeg; };
virtual    int      isEqual( const ClasseBase& ) const
                         { return 0; };

const      int      eval( ListeLineaire * = NULL )
                         { return ERROR;};

virtual    int      adresse( ) const = 0;

virtual    char *nom( ) {return n; };

     int&          operator() ( int i );
     int           operator() ( int i ) const;

virtual    int      BorneInf( ) const = 0;
virtual    int      BorneSup( ) const = 0;
};
```

Il ne peut pas exister d'instance de cette classe, car certaines méthodes ne peuvent pas être définies pour cette classe mais doivent l'être dans les classes dérivées : il s'agit de l'adresse du tableau ou du segment, des bornes.

Méthodes :

ClasseTabSeg

Ce constructeur initialise la chaîne de caractères contenant le nom du tableau ou segment.

isA

Identifiant de la classe **ClasseTabSeg.**

isEqual

Il faut déclarer cette fonction, car elle est déclarée dans la classe de base, mais elle se sera jamais utilisée puisqu'il ne peut exister aucune instance de cette classe.

eval

Il faut déclarer cette fonction, car elle est déclarée dans la classe de base, mais elle se sera jamais utilisée puisqu'il ne peut exister aucune instance de cette classe.

adresse

Cette fonction retourne l'adresse permettant d'avoir accès aux éléments du tableau ou du segment. Dans le cas du tableau, il s'agit de l'adresse qui lui a été allouée, dans le cas d'un segment, il s'agit de l'adresse du tableau qui lui est associé. Cette adresse est utilisée pour les fonctions permettant d'avoir accès à un élément du tableau ou du segment définies ci-dessous.

nom

Retourne le nom du tableau ou du segment.

operator()

Cet opérateur permet d'accéder aux éléments du tableau ou du segment. La première déclaration retourne l'adresse de cet élément pour permettre une mise-à-jour de la donnée.
Si 't' est un tableau ou un segment, on peut écrire

```
t(1) = 5;
```

comme on écrirait

```
t[1] = 5;
```

pour un tableau standard en C.

La deuxième déclaration retourne simplement la valeur stockée à l'indice i dans la tableu ou le segment.
Si 't' est un tableau ou un segment, on peut écrire

```
if( t(1) == 5 ) ... ;
```

comme on écrirait

```
if( t[1] == 5 ) ... ;
```

pour un tableau standard en C.

Ces deux méthodes utilisent la méthode **adresse** pour savoir où trouver l'élément souhaité.

BorneInf

Cette fonction retourne la valeur de la borne inférieure du tableau ou du segment. Comme cette valeur dépend du fait que l'on ait un tableau ou un segment, la définition de cette fonction est placée dans les classes des tableaux et des segments.

BorneSup

Cette fonction retourne la valeur de la borne supérieure du tableau ou du segment. Comme cette valeur dépend du fait que l'on ait un tableau ou un segment, la définition de cette fonction est placée dans les classes des tableaux et des segments.

Classe ClasseTableau

Cette classe va reprendre tous ce qui est propre aux tableaux : adresse mémoire et borne inférieure et supérieure.

```
class ClasseTableau : public ClasseTabSeg {
protected:
    int adr, bi, bs;

public:
            ClasseTableau( const char *q = NULL ) :
                ClasseTabSeg( q ) {};
            ~ClasseTableau( );

virtual   int       isA( ) const { return _ClasseTableau; };
virtual   int       isEqual( const ClasseBase& ) const;

ClasseTableau&      operator=( const ClasseTableau& t );

          void      adresse( int _adr, int _bi, int _bs );
          int       adresse( ) const { return adr; };
```

```
        int         BorneInf( ) const;
        int         BorneSup( ) const;
};
```

Méthodes :

ClasseTableau

Ce constructeur ne fait que nommer le tableau en appelant le constructeur de la classe **ClasseTabSeg.**

isA

Retourne l'identifiant de la classe.

isEqual

Vérifie si l'argument est bien l'occurence de la classe, c'est-à-dire s'il s'agit d'un tableau, s'il porte le même nom et s'il possède les mêmes valeurs de borne.

operator=

Cet opérateur permet d'affecter les éléments du tableau donné en argument à l'occurence courante du tableau. Si le nombre d'éléments n'est pas identique dans les deux tableaux, rien n'est modifié.

adresse

Cette méthode a deux définitions :
- la première permet de stocker l'adresse et les bornes du tableau pour définir celui-ci;
- la deuxième permet d'obtenir l'adresse du tableau, qui a été stockée via la définition ci-dessus.

BorneInf

Retourne la valeur de la borne inférieure du tableau.

BorneSup

Retourne la valeur de la borne supérieure du tableau.

Classe ClasseSegment

Un segment est une partie de tableau dont les bornes sont délimitées par des expressions arithmétiques.

```
class ClasseSegment : public ClasseTabSeg {
protected:
     ClasseTableau *tab;
public:
     ClasseExpression bi, bs;

                ClasseSegment( ClasseTableau *t = NULL,
                     const char *p = NULL,
                     ClasseExpression *_bi = NULL,
                     ClasseExpression *_bs = NULL );
                ~ClasseSegment( );

virtual    int  isA( ) const { return _ClasseSegment; };
virtual    int  isEqual( const ClasseBase& ) const;

ClasseSegment&       operator=( const ClasseSegment& t );

           int  BorneInf( ) const;
           int  BorneSup( ) const;

ClasseTableau  *tableau( ) const { return tab; };

virtual    int  adresse( ) const { return tab->adresse( ); };
};
```

Méthodes :

isA

Retourne l'identifiant de la classe.

isEqual

Vérifie si l'argument est bien le segment que l'on souhaite retrouver.

ClasseSegment

Constructeur du segment avec les différents arguments nécessaire (nom, tableau, bornes).

~ClasseSegment

Destructeur de la classe. Cette méthode va détruire les expressions.

operator=

Opérateur permettant d'initialiser le segment courant avec les éléments se trouvant dans le segment fourni en argument.

BorneInf

Retourne la valeur de la borne inférieure du segment. Cette valeur est obtenue par l'évaluation de l'expression associée représentant cette borne.

BorneSup

Retourne la valeur de la borne supérieure du segment. Cette valeur est obtenue par l'évaluation de l'expression associée représentant cette borne.

tableau

Retourne l'adresse de l'occurence de la classe **ClasseTableau** représentant le tableau dans lequel le segment est inscrit.

adresse

Retourne l'adresse du premier élément du tableau dans lequel le segment est inscrit.

Classe ClasseVarInd

Cette classe permet d'avoit accès à un élément d'un tableau ou d'un segment.

```
class ClasseVarInd : public ClasseBase {
protected:
     ClasseTabSeg *tab;

public:
                ClasseVarInd( ClasseTabSeg *t = NULL );
                ~ClasseVarInd( );

virtual   int   isA( ) const { return _ClasseVarInd; };

ClasseVarInd&   operator=( const ClasseVarInd& t );

const     int   eval( ListeLineaire * = NULL );
};
```

Méthodes :

isA

Retourne l'identifiant de la classe.

ClasseVarInd

Le constructeur de la classe prend comme argument le tableau ou segment dans lequel il faut aller chercher la donnée.

eval

Place sur la pile la valeur contenue dans un des éléments du tableau ou du segment. L'indice de l'élément dans le tableau se trouve sur le sommet de la pile. La valeur du tableau ou segment sera mise à la place de cet indice.

8. Les expressions

La syntaxe des expressions est définie dans le premier chapitre. Une expression sera du type logique si elle possède un opérateur ou une fonction logique, sinon elle est de type arithmétique.

L'évaluation d'une expression arithmétique retourne la valeur de cette expression , tandis qu'une expression logique retourne 1 si elle est évaluée à vrai, et 0 si elle vaut faux. Les évaluations sont régies par la sémantique des opérateurs et des fonctions définies dans le chapitre 1.

Une expression est représentée sous le format postfixé et est placée dans une liste linéaire.

La classe **ClasseExpression** permet de réaliser les évaluations des expressions.

Les arguments des opérateurs sont placés et évalués avant l'opérateur. Pour les fonctions, les arguments se trouvent après l'occurence de la fonction. Comme le nombre d'argument est fixe, on peut trouver rapidement les arguments que la fonction doit utiliser. Seulement, certains arguments doivent être évalués comme pour 'somme(head(a))' qui calcule la somme de début du tableau ou segment a, ou comme si on utilise un segment variable du style 'a{i:j}'. L'évaluation de ces arguments sera exécutée avant l'évaluation de la fonction.

```
class ClasseExpression : public ClasseBase {
private:
     ListeLineaire *l;
     char n[MAXVARLEN + 1];
public:
          ClasseExpression( ListeLineaire *liste = NULL,
                    char *nom = NULL );
          ~ClasseExpression( ) { if( l ) l->detruire( ); };

virtual   int  isA( ) const { return _ClasseExpression; };
virtual   int  isEqual( const ClasseBase& ) const;

ClasseExpression&   operator=( const ClasseExpression& t )
               {
                    *l = *(t.l);
                    return *this;
               };

const     int  eval( ListeLineaire * = NULL );

ListeLineaire  *Liste( ) const { return l; };

virtual   char *nom( ) { return n; };
          void nom( const char *p ) { strcpy( n, p ); };
};
```

Méthodes :

ClasseExpression

Ce constructeur définit une expression grâce à la liste linéaire qui est fournie en argument. Il est possible de donner un nom à une expression, mais ce n'est pas obligatoire. En effet, les expressions de bornes de segments n'ont pas besoin de nom, tandis que les expressions déclarées par l'utilisateur doivent posséder un nom.

~ClasseExpression

Ce destructeur va détruire la liste linéaire qui a contenu l'expression.

isA

Retourne l'identifiant de la classe.

isEqual

Vérifie que l'expression qui est donnée en argument est égale à l'occurence courante de la classe. La fonction retourne 1 si les nom sont identiques et si tous les éléments de la liste linéaire sont identiques et dans le même ordre, sinon retourne 0.

operator=

Permet d'affecter l'expression donnée en argument à l'occurence courante de la classe.

eval

L'évaluation d'une expression revient à évaluer tous les éléments de la liste linéaire. Les méthodes '**eval**' des différentes classes définies ci-dessus seront appelées pour évaluer les différentes éléments de l'expression.

Liste

Cette fonction retourne l'adresse de la liste linéaire contenant la représentation de l'expression.

nom

Si cette fonction est appelée avec un argument, elle permet de donner un nom à l'expression. Sinon, elle retourne le nom de l'expression.

9. Les instructions.

Les différentes instructions du programme définies dans le langage sont placées dans un tableau. Il y a un maximum de **MAXINSTRUCTION** instructions. Il peut y voir **MAXLABEL** étiquettes différentes.

Les instructions sont représentées sous forme de 'record' et d''union' C. Chaque élément du tableau de programme pourra recevoir une des représentations des instructions.

Condition : if

Pour une instruction de condition, il faut juste conserver l'adresse de l'expression logique et le numéro de l'étiquette où poursuivre l'exécution du programme si le résultat de l'évaluation de cette expression est vrai.

```
typedef struct ins_si {
CLasseExpression *b;
unsigned adr;
} ins_si;
```

Saut inconditionnel : goto

Pour cette instruction, il faut juste connaître le numéro de l'étiquette où il faut se rendre pour poursuivre l'exécution du programme.

```
typedef struct ins_goto {
unsigned adr;
} ins_goto;
```

Evaluation : eval

Pour cette instruction, il faut connaître l'adresse de la variable ou de l'expression dont il faut évaluer la valeur.
Comme on peut avoir une variable ou une expression, on doit définir le champ de la structure comme un pointeur vers la classe de base.

```
typedef struct ins_eval {
ClasseBase *o;
} ins_eval;
```

Affectation : let

Pour affecter une valeur à une variable entière ou logique, on a besoin de l'adresse de la variable et de l'expression permettant de calculer la valeur.

Pour affecter une valeur à un élément d'un tableau ou d'un segment, on a besoin de l'adresse du tableau ou du segment, de l'expression permettant de calculer la valeur de l'indice et de l'adresse de l'expression permettant de calculer la valeur à stocker.

Le champ permettant d'avoir accès à la variable ou au tableau est défini comme un pointeur vers la classe de base. Le champ **e1**, si il existe, définit l'expression de l'indice et e2 définit l'expression de la valeur à placer en mémoire.

```
typedef struct ins_let {
    ClasseBase *v;
    ClasseExpression *e1, *e2;
} ins_let;
```

Impression de texte : print

Il faut juste enregistrer le texte à imprimer.

```
typedef struct ins_pri {
    char *p;
} ins_pri;
```

Impression de variable : printvar

Pour imprimer la valeur d'une variable entière ou logique, on a besoin de l'adresse de la variable.

Pour imprimer la valeur à un élément d'un tableau ou d'un segment, on a besoin de l'adresse du tableau ou du segment et de l'expression permettant de calculer la valeur de l'indice .

Le champ permettant d'avoir accès à la variable ou au tableau est défini comme un pointeur vers la classe de base. Le champ **e1** définit l'expression de l'indice.

```
typedef struct ins_priv{
    ClasseBase *v;
    ClasseExpression *e1;
} ins_priv;
```

Passer à la ligne : printnl

Il n'y a aucune information à conserver pour cette instruction.

Lecture d'une donnée : read

Les données nécessaire pour lire une valeur au clavier et l'affecter à une variable ou à un élément d'un tableau ou segment sont identiques aux données de l'instruction **printvar**. Cette structure sera

Instruction d'arrêt : stop

Cette instruction n'a pas besoin de donnée.

Instruction d'exécution : run

Cette instruction n'a pas de représentation interne : elle commence l'évaluation du programme à partir de la première instruction.

Une structure d''union' a été réalisée avec toutes les données ci-dessus. Cette union et une valeur entière indiquant le type d'instruction forment la représentation des instructions dans le tableau **program** qui contient le programme à exécuter.

```
typedef union instruction {
     ins_si si;
     ins_goto go;
     ins_eval eval;
     ins_let let;
     ins_pri pri;
     ins_priv priv;
} instruction;


typedef struct prog {
     int genre;
     instruction i;
} prog;


#define MAXINSTRUCTION 1000

prog program[ MAXINSTRUCTION ];
unsigned lastins = 0;
```

La variable **lastins** indique la position où l'on pourra placer la prochaine instruction.

Pour les étiquettes, on a un tableau dont chaque élément est le numéro d'une étiquette et son adresse dans le programme.

```
typedef struct etiquette {
     unsigned num;
     unsigned adr;
} etiquette;


#define MAXLABEL 100

etiquette label[ MAXLABEL ];
unsigned lastlab = 0;

unsigned cour_inst;
int running;
```

La variable **lastlab** indique le nombre d'étiquettes définies dans le programme.

La variable **cour_inst** indique le numéro de l'instruction courante dans le programme.

La variable **running** indique si on est en train d'exécuter le programme. Si elle vaut 1, on exécute le programme, sinon elle vaut 0.

Il y a plusieurs procédures pour placer ces instructions dans le tableau et pour exécuter leurs évaluations.

```
int ajo_adr( int eti );
```

Cette instruction ajoute l'étiquette donnée en argument dans le tableau **label** et l'associe à l'adresse **lastins**.

```
int adr_eti( int eti );
```

Retourne l'adresse associée à l'étiquette fournie en argument. Si l'étiquette n'est pas définie dans le tableau **label**, la valeur -1 est retournée.

```
int ajo_inst_si( ClasseExpression *b, unsigned adr );
```

Ajoute une instruction de saut conditionnel à l'adresse **lastins**. **b** est l'adresse de l'expression qui doit être vérifiée, **adr** est le numéro de l'étiquette où doit se poursuivre l'exécution du programme si **b** est évaluée à vrai.

```
int ajo_inst_goto( unsigned adr );
```

Ajoute une instruction de saut inconditionnel à l'adresse **lastins**. **adr** est le numéro de l'étiquette où doit se poursuivre l'exécution du programme.

```
int ajo_inst_eval( ClasseBase *o );
```

Ajoute une instruction d'évaluation de variable ou d'expression à l'adresse **lastins**. **o** est l'adresse de la variale ou de l'expression.

```
int ajo_inst_let( ClasseBase *v, ClasseExpression *e1,
                  ClasseExpression *e2 );
```

Ajoute une instruction d'affectation à l'adresse **lastins**. **v** est l'adresse de la variable qui doit recevoir la valeur, ou l'adresse du tableau ou segment s'il s'agit d'une variable indicée. Si **e1** n'est pas l'adresse **NULL**, e1 représente l'adresse de l'expression de l'indice dans le cas d'une variable indicée. **e2** est l'adresse de l'expression dont l'évaluation fournit la valeur à affecter à la variable.

```
int ajo_inst_stop( void );
```

Ajoute une instruction d'arrêt à l'adresse **lastins** du programme.

```
int ajo_inst_pri( const char *p );
```

Ajoute une instruction d'impression de texte. On donne le texte à imprimer comme argument. Une copie de ce texte est effectuée et c'est cette copie qui est conservée par l'instruction.

```
int ajo_inst_priv( ClasseBase *v, ClasseExpression *e1 );
```

Ajoute une instruction d'impression d'une variable à l'adresse **lastins** du programme. **v** représente l'adresse de la variable ou du tableau (segment). Si **e1** n'est pas l'adresse **NULL**, e1 représente l'adresse de l'expression dont l'évaluation fournit l'indice de l'élément à imprimer dans le tableau (segment).

```
int ajo_inst_read( ClasseBase *v, ClasseExpression *e1 );
```

Ajoute une instruction de lecture d'une variable à l'adresse **lastins** du programme. **v** représente l'adresse de la

variable ou du tableau (segment). Si **e1** n'est pas l'adresse **NULL**, e1 représente l'adresse de l'expression dont l'évaluation fournit l'indice de l'élément à imprimer dans le tableau (segment).

```
int ajo_inst_flush( void );
```

Ajoute une instruction vidant les données se trouvant sur l'entrée (clavier) à l'adresse **lastins** du programme.

```
int ajo_inst_nl( );
```

Ajoute une instruction passant à la ligne sur la sortie standard à l'adresse **lastins** du programme.

Toutes les fonctions ci-dessus retournent 1 si on a pu ajouter l'instruction, 0 sinon. Si on a ajouté l'instruction, l'adresse **lastins** est incrémentée de 1.

Pour exécuter le programme, on appelle la fonction suivante qui va evaluer les instructions.

```
void run_program( );
```

Cette fonction utilise la fonction **eval_inst** qui évalue l'instruction qu'on lui donne en argument. Cette fonction est consitutée d'un 'case' en fonction du type de l'instruction à évaluer.

Lors de l'évaluation des instructions, il faut évaluer des expressions, des variables, ... Ces évaluations placent le résultat sur la pile des valeurs. La fonction suivante récupère le résultat de la dernière évaluation et le retire de la pile.

```
int resultat( void );
```

10. Traduction.

La traduction des déclarations et du programme se déroule en une seule passe. C'est pour cette raison que tous les éléments doivent être déclarés avant d'être utilisé (sauf pour les étiquettes).

Le traducteur est composé de deux parties principales : la traduction des variables et des expressions, et la traduction des instructions.

La traduction des expressions est effectuée par la fonction

char *traduction(ListeLineaire *trad, char *s, char term)

Cette fonction place à la fin de la liste linéaire, dont l'adresse est fournie par **trad**, la traduction de la chaîne **s** qui doit se terminer par le caractère contenu dans **term**. La fonction retourne la chaîne **s** qu'il reste à traduire.

Cette fonction utilise une pile locale, qui enregistre les opérateurs. Dans le bas de la pile se trouvent les opérateurs de plus faibles priorités.

Algorithme

```
vider( pile );

A:

lire S

si S est fct alors
     traduire S et arguments
sinon
     si S est variable
     traduire S
     sinon
          si S est nombre
          traduire nombre
          sinon
               si S = '(' alors
                    récurrence à partir du caractère suivant
                    '(' et jusqu'au caractère ')' correspondant
               sinon erreur

lire S

D:
si S est operateur alors
     p := priorité( S )
sinon
     si p <> term alors erreur
     sinon p := 0
```

```
B:

si  la pile est vide  alors aller en C

si priorité( sommet pile) < p alors aller en C

traduire operateur sur le sommet de la pile
enlever opérateur sur le sommet de la pile

aller en B

C:

si p <> 0 alors
     empiler opérateur S
     aller en A


fin
```

L'opérateur **NOT** est particulier dans cet algorithme. En effet, on vérifie si on a une fonction ou une variable. Or l'argument de cet opérateur est situé après l'opérateur. Pour résoudre cela, j'ai placé l'opérateur comme une fonction, mais la traduction de cette fonction sera un saut juste avant le test pour vérifier si on a un opérateur, c'est-à-dire à l'adresse D dans l'algorithme.

Pour la déclaration des variables, tableaux et segments, il y a une fonction C qui permet de créer l'élément souhaité et en initialisant les champs dans la classe. Ces fonctions retournent l'adresse de l'élément qui vient d'être créé et elles le placent dans l'environnement.

```
ClasseVarEntiere *allouer_var_ent( const char *nom );

ClasseVarLogique *allouer_var_log( const char *nom );

ClasseTableau *allouer_tab( const char *nom, int bi, int bs );

ClasseSegment *allouer_seg( ClasseTableau *t, const char *nom,
                    ClasseExpression *_bi,
                    ClasseExpression *_bs );
```

Ces fonctions sont utilisées conjointement avec la fonction suivante pour éviter que l'on crée un objet ayant le même nom qu'un élément se trouvant dans l'environnement.

Cette fonction va retourner l'adresse de l'élément dont le nom en fournit en paramètre. Si un tel élément n'existe pas, l'adresse **NULL** est retournée. Grâce à cette fonction, le traducteur peut vérifier si un identificateur existe déjà dans l'environnement, et si non, ajouter cet élément dans l'environnement grâce aux fonctions ci-dessus.

```
ClasseBase *dans_env( const char *p );
```

11. Conclusion

Ce programme a été developpé au moyen du langage orienté objet, le C++. Ce langage forunit des avantages comme nous l'ovons décrit au début du chapitre.

Ces avantages permettent-ils un réel gain de temps de programmation ?

Nous en sommes convaincus et ce, pour les raisons développées ci-dessous.

En premier lieu, ce langage apporte la notion de classe. Bien qu'identique à une structure en C, il peut donner des identificateurs de fonctions dans cette structure. Ces fonctions pourront être utilisées uniquement par une instance de la classe correspondante. Grâce à cette structure, on peut cacher les variables à l'utilisateur (principe de l'encapsulation) et autoriser l'accès aux valeurs via les fonctions déclarées. Cette procédure interdit des modifications des variables par des fonctions non autorisées. En effet, toutes les variables d'une classe ne peuvent être modifiée que par une fonction de classe. Une autre méthode possible de modification est autorisée en permettant à une classe dérivée (et à ses fonctions) d'avoir accès à une (ou plusieurs) variable(s) ou à toutes ses fonctions.

Un autre avantage du C++ et de la notion de classe est l'héritage. On peut définir une classe A et B, en déclarant que B hérite des propriétés de A. Cela veut dire que B peut avoir accès à toutes les fonctions définies dans A, aux variables, etc ... . Cette notion d'héritage permet de définir une classe standard en reprennant les concepts communs à plusieurs objets et de définir les fonctions qui manipulent ces concepts. Il ne reste plus qu'à déclarer les classes qui correspondent aux différents objets en leurs donnant l'héritage de cette classe commune. On peut également définir un concept commun à chaque classe, exigeant un algorithme différent pour rechercher le résultat.

Dans ce mémoire, la classe **ClasseTabSeg** reprend les notions communes aux tableaux et aux segments. La différence entre les deux, est que le tableau occupe une place réelle dans la mémoire du programme, alors que le segment est unerégion d'un tableau. Il est possible de définir des segments qui recouvrent des zones communes, mais il est impossible d'agir de même pour les tableaux. Un segment sera donc toujours associé à un tableau. Pour avoir accès à un élément d'un tableau ou d'un segment, il faut connaître l'adresse dans la mémoire du programme. Celle-ci se trouve grâce à l'adresse du tableau et de l'indice de l'élément. Dans le cas d'un tableau, on connaît tout de suite son adresse, tandis que pour un segment, il faut aller voir, dans le tableau qui lui est associé, l'adresse nécessaire. Dans la classe **ClasseTabSeg**, on trouvera donc une fonction qui retourne l'adresse du premier élément d'un tableau ou d'un segment.

La fonction permettant d'avoir accès à un élément d'un tableau ou d'un segment sera donc commune aux deux classes, mais elle utilisera la fonction retournant l'adresse du premier élément, qui elle est définie dans chaque classe dérivée. La liaison dynamique permet de choisir la fonction à

utiliser (tableau ou segment) lors de l'exécution du programme.

Malgré la liaison dynamique qui ralentit l'exécution du programme, nous pensons que l'utilisation du C++ facilite la programmation. En effet, cette liaison dynamique joue le rôle d'un 'switch' en C, en exécutant la fonction associée à l'instance de classe que l'on utilise. Dans le cas des expressions, l'evaluation de celle-ci n'est que l'evaluation des instances de toutes les classes qui la compose. Comme chaque type d'objet est déclaré dans une classe dérivée de la classe de base qui reprend les différentes fonctions qui se retrouvent dans TOUTES les classes, la fonction d'évaluation s'y retrouve également. On peut donc appeler cette fonction grâce à un pointeur vers la classe de base, et C++ utilisera la fonction d'évaluation de la classe adéquate.

Il est donc possible d' ajouter un nouvel objet à l'ensemble du système, et si les fonctions de la classe de base sont toutes reprises et redéfinies correctement pour la nouvelle classe, il n'est pas nécessaire de modifier les autres classes, comme les expressions. La fonction d'évaluation de la nouvelle classe sera appelée grâce à la liaison dynamique si elle est reprise dans une expression. Grâce à la liaison dynamique, on ne doit plus définir de 'switch' explicitement : on va donc éviter une mauvaise mise-à-jour du programme en oubliant d'ajouter une option à tel ou tel switch.

Chapitre V. Extensions / améliorations.

- Le jeu des instructions du langage ne reprend que des instructions simples. On peut le comparer à l'assembleur des ordinateurs. Il serait interessant d'augmenter le jeu des instructions pour arriver à un système similaire au Pascal, où l'on trouve les boucles 'whiles' et 'repeat', les conditions, les affectations, ... On peut, soit définir un macro-compilateur qui fournit un code n'utilisant que les instructions définies ci-dessus, soit ajouter simplement les instructions ou les remplacer par des nouvelles.
  Le macro-compilateur permet en plus de montrer aux étudiants la traduction d'instructions complexes (while, repeat, ...) en instructions de base.

- Il serait interéssant que certaines fonctions soient étendues aux variables (minimum, maximum) et puissent prendre un nombre quelconque d'arguments.

- On peut imaginer de changer la gestion des arguments pour les fonctions en placant les arguments ailleurs que dans l'expressions.

- La description graphique des situations a été défini par Jeannine Rulkin, mais la réalisation pourrait faire l'objet d'un autre mémoire. Il est souvent plus facile d'exprimer les conditions quand on se les représente avec un dessin.

- Il reste à implémenter les segments variables, c'est-à-dire les egments dont les bornes sont limitées par une valeur minimale et maximale. Il paraît bien évident que la méthode de description graphique proposée par J. Rulkin est plus rapide et mieux adaptée à cette résolution de problème. Il est possible de simuler ces segments grâce à l'opérateur 'pourtout' :

  pourtout a 1 : n : pourtout b a : n : somme(tab[a:b])

  va calculer toutes les sommes de tous les segments que l'on peut construire dans le tableau 'tab' entre 1 et n.

- Il faudrait améliorer la configuration du système (taille de la mémoire des variables et tableaux, tailles des programmes, nombres d'étiquette) : ne pas être obligé de recompiler le programme lorsque l'on change les valeurs.

- Enfin, une amélioration devrait encore fournir une optimisation des constantes : si une constante revient deux fois, utiliser la même occurence et non deux occurences différentes.

BIBLIOGRAPHIE

M. Derroitte et B. Le Charlier,
Un système d'aide à l'enseignemnt d'une méthode de programmation,
Institut d'Informatique, Namur, Mai 1988

D. Fisette,
Définition d'un langage de programmation permettant l'expression d'assertion,
mémoire de maîtrisse, Namur 1985.

B. Le Charlier,
Définition du langage LSD80,
Institut d'Informatique, Namur, Septembre 1980

Notes de cours de B. Le Charlier,
Preuves de programmes,
Institut d'Informatique, Namur

J. Rulkin,
Contribution à l'implémentation d'un langage graphique de description d'assertions,
Mémoire de maîtrisse, Namur 1989

REALISATION D'UN PROGRAMME
DE VERIFICATION
DES INVARIANTS

Annexe

Luc SIMON

Mémoire réalisé sous la
direction du Professeur B. Le Charlier
en vue de l'obtention du titre de
Licencié et Maître en Informatique.

Namur — Septembre 1992

Fichier CCONST.HPP

Fichier des constantes

```
#ifndef __IOSTREAM_H
#include <iostream.h>
#endif


#ifndef __CCONST_HPP
#define __CCONST_HPP

#define MAXPILEVAL              100


#define _ClasseBase             1

#define _ClasseListeLin         2

#define _ClasseVariable         3
#define _ClasseVarEntiere       4
#define _ClasseVarLogique       5

#define _ClasseConstante        7

#define _ClasseTabSeg           8
#define _ClasseTableau          9
#define _ClasseVarInd           10
#define _ClasseSegment          11

#define _ClasseFonction         16
#define _ClassePlus             17
#define _ClasseMoins            18
#define _ClasseFois             19
#define _ClasseDiv              20
#define _ClasseMod              21
#define _ClasseNeg              22
#define _ClasseSomme            23
#define _ClasseProduit          24
#define _ClasseMin              25
#define _ClasseMax              26
#define _ClasseLong             27
#define _ClasseFirst            28
#define _ClasseLast             29
#define _ClasseEmpty            30
#define _ClasseTriCroi          31
#define _ClasseTriDec           32
#define _ClasseTriStrCroi       33
#define _ClasseTriStrDec        34
#define _ClassePair             35
#define _ClasseInchange         36
#define _ClasseExamine          37
#define _ClassePermute          38

#define _ClasseIsSegment        39
#define _ClassePrefixe          40
#define _ClasseSuffixe          41
#define _ClasseEgal             42
#define _ClassePermutation      43

#define _ClasseConcat           44

#define _ClasseAnd              45
#define _ClasseOr               46
#define _ClasseNot              47
#define _ClasseEq               48
#define _ClasseNe               49
```

```
#define _ClasseGe               50
#define _ClasseGt               51
#define _ClasseLe               52
#define _ClasseLt               53
#define _ClasseHead             54
#define _ClasseTail             55

#define _ClasseExpression       56

#define _ClassePourTout         57
#define _ClasseIlExiste         58



#define MAXVARLEN                   32

/*
  Les deux constantes ci-dessous définissent deux entiers particuliers
 :
  'ENT_NON_DEF' représente un entier non défini, tant dis que
  'OVERFLOW' représente la sortie de l'ensemble des valeurs du type en
tier.
  Les valeurs utilisables vont donc de -32767 à 32766.
  ERROR signale une erreur lors de l'évaluation d'une classe.
*/

#define ENT_NON_DEF             -32768
#define OVERFLOW                 32767

#define ERROR                    32765

/*
  L'intervalle des valeurs utilisables pour les calculs va de MINVAL à
 MAXVAL
  inclus. Les autres valeurs sont utilisées ci-dessus.
*/
#define MAXVAL                   32500
#define MINVAL                  -32500

/*
 */

#define EXAMINE          1
#define PERMUTE          2

/*
   Définition en avant des classes utilisées.
*/
class far ClasseBase;
class far ListeLineaire;

class far ClasseVariable;
class far ClasseVarEntiere;
class far ClasseVarLogique;
class far ClasseVarBoucle;

class far ClasseConstante;

class far ClasseTabSeg;
class far ClasseTableau;
class far ClasseVarInd;
class far ClasseSegment;
class far ClasseSegmentFF;
```

```
class far ClasseSegmentFV;
class far ClasseSegmentVF;
class far ClasseSegmentVV;

class far ClasseFonction;
class far ClassePlus;
class far ClasseMoins;
class far ClasseFois;
class far ClasseDiv;
class far ClasseMod;
class far ClasseNeg;
class far ClasseSomme;
class far ClasseProduit;
class far ClasseMin;
class far ClasseMax;
class far ClasseLong;
class far ClasseFirst;
class far ClasseLast;
class far ClasseEmpty;
class far ClasseConcat;
class far ClasseEgal;
class far ClassePermutation;
class far ClasseTriCroi;
class far ClasseTriDec;
class far ClasseTriStrCroi;
class far ClasseTriStrDec;
class far ClassePair;
class far ClasseIsSegment;
class far ClassePrefixe;
class far ClasseSuffixe;
class far ClasseInchange;
class far ClasseExamine;
class far ClassePermute;
class far ClasseAnd;
class far ClasseOr;
class far ClasseNot;
class far ClasseEq;
class far ClasseNe;
class far ClasseGe;
class far ClasseGt;
class far ClasseLe;
class far ClasseLt;
class far ClasseHead;
class far ClasseTail;

class far ClasseExpression;

class far ClassePourTout;
class far ClasseIlExiste;


#endif
```

Fichier CBASE.HPP

Classe de base

```
#ifndef __IOSTREAM_H
#include <iostream.h>
#endif

#include <stdio.h>

#include "cconst.hpp"

/*
 'env' représente l'environnement du programme : ce tableau contient l
es
 différentes variables qui sont déclarées dans le programme, les table
aux,
 les segments, ...
 On accède à un élément grâce à une clé obtenue par une fonction de 'h
ashage'
 */

extern ListeLineaire env;

/*
 'mem' est la 'mémoire' du programme. C'est dans ce 'tableau' que sont
 conservées les valeurs des variables. 'nadr' est l'adresse de la proc
haine
 place libre pour allouer de la mémoire.
 'mem' est un pointeur qui est utilisé comme tableau : le i ème élémen
t est
 à l'adresse 'mem + i'.
 */
extern int *mem, *mem0, *exa;
extern int nadr;

extern long pile_val[MAXPILEVAL];
extern int  sommet_pile_val;
int resultat( void );

extern int NextSegment;


#ifndef __MEM_HPP
#define __MEM_HPP
/*
 'init_mem' est une fonction pour initialiser les variables ci-dessus.
 */
void init_mem( );

/*
 'term_mem' détruit le 'tableau' ci-dessus.
 */
void term_mem( );

/*
 * La fonction 'allouer_var' va reserver de la place pour une variable
 * entière ou booléenne. Si l'allocation est possible, elle retourne 1
,
 * sinon elle retourne 0.
 */
ClasseVarEntiere *allouer_var_ent( const char *nom );
ClasseVarLogique *allouer_var_log( const char *nom );

/*
 * La fonction 'allouer_tab' va reserver de la place pour un tableau d
'entiers
 * ou de valeurs logiques de 'bs - bi + 1' éléments. 'bi' est l'indice
```

```
 minimum
 * et 'bs' est l'indice maximum. Si l'allocation est possible, elle re
tourne
 * 1, sinon elle retourne 0.
 */
ClasseTableau *allouer_tab( const char *nom, int bi, int bs );


ClasseSegment *allouer_seg( ClasseTableau *t, const char *nom,
                            ClasseExpression *_bi, ClasseExpression *_
bs );

int lire_mem( int adr );

void ecrire_mem( int adr, int val );



void init_mem0( );

void term_mem0( );

int lire_mem0( int adr );

ClasseBase *dans_env( const char *p );
#endif



#ifndef __CBASE_HPP

#define __CBASE_HPP

class far ClasseBase {
protected:
        int     err;

public:
                        ClasseBase( ) {};
                       ~ClasseBase( ) {};

virtual int             isA( ) const = 0;
virtual int             isEqual( const ClasseBase& ) const = 0;

        ClasseBase&     operator=( const ClasseBase& ) {return *this;}
;

virtual const int       eval( ListeLineaire *l = NULL ) = 0;
        int             resultat( )
                              { return (int)pile_val[ --sommet_pile_va
l ]; };

        int             error( ) const { return err; };
        void            error( int v ) { err = v; };

virtual char           *nom( ) { return ""; };
        void            nom( char *) { };

};

#endif
```

Fichiers EXPR.HPP et EXPR.CPP

Fichiers des expressions

```
#include <mem.h>
#include <string.h>

#ifndef __IOSTREAM_H
#include <iostream.h>
#endif


#ifndef __CCONST_HPP
#include "cconst.hpp"
#endif


#ifndef __CBASE_HPP
#include "cbase.hpp"
#endif


#ifndef __LISTELIN_HPP
#include "ListeLin.hpp"
#endif



#ifndef __CEXPR_HPP
#define __CEXPR_HPP


class ClasseExpression : public ClasseBase {
private:
        ListeLineaire *l;
        char n[MAXVARLEN + 1];
public:
                        ClasseExpression( ListeLineaire *liste = NULL,
 char *nom = NULL );
                       ~ClasseExpression( ) { if( l ) l->detruire( );
};

virtual int     isA( ) const { return _ClasseExpression; };
virtual int              isEqual( const ClasseBase& ) const;

        ClasseExpression&       operator=( const ClasseExpression& t )
                        {
                          *l = *(t.l);
                          return *this;
                        };

const   int              eval( ListeLineaire * = NULL );
        int              resultat( )
                         { return (int)pile_val[ sommet_pile_val-- ]; }
;

        ListeLineaire  *Liste( ) const { return l; };

virtual char            *nom( ) { return n; };
        void             nom( const char *p ) { strcpy( n, p ); };

};
#endif
```

```
#include <stdlib.h>
#include <string.h>

#include "expr.hpp"

ClasseExpression::ClasseExpression( ListeLineaire *liste, char *nom )
{
  l = new ListeLineaire( );
  l->creer( );
  if( liste ) *l = *liste;
  if( nom ) strcpy( n, nom );
  else strcpy( n, "" );
};


int     ClasseExpression::isEqual( const ClasseBase& b ) const
{
  int v;
  ClasseBase *b1, *b2;
  ListeLineaire *e;

  if( !strcmp( n, "" ) && !strcmp( ((ClasseExpression&)b).n, "" ) ) re
turn !strcmp( n, ((ClasseExpression&)b).n );
  e = ((ClasseExpression&)b).Liste( );
  v = 1;
  if( l->nbr( ) != e->nbr( ) ) return 0;
  l->repositionner( );
  e->repositionner( );

  while( !l->eof( ) && v )
  {
    b1 = l->courant( );
    b2 = e->courant( );
    v = b1->isEqual( *b2 );
    l->avancer( );
    e->avancer( );
  };

  return v;
}

const int       ClasseExpression::eval( ListeLineaire *l2 )
{
  ClasseBase *b;
  int v;

  l->repositionner( );
  v = 1;
  while( !l->eof( ) && v )
  {
    b = l->courant( );
    l->avancer( );
    v = ( b->eval( l ) == 0 );
  };

  if( v ) return 0;
  else return ERROR;
}
```

Fichiers FCT.HPP et FCT.CPP

Fichiers des fonctions et
des opérateurs 'pour tout'
et 'il existe'

```
#include <mem.h>
#include <string.h>

#ifndef __IOSTREAM_H
#include <iostream.h>
#endif


#ifndef __CCONST_HPP
#include "cconst.hpp"
#endif


#ifndef __CBASE_HPP
#include "cbase.hpp"
#endif


#ifndef __LISTELIN_HPP
#include "ListeLin.hpp"
#endif

#ifndef __VAR_HPP
#include "var.hpp"
#endif

#ifndef __EXPR_HPP
#include "expr.hpp"
#endif


#ifndef __FCT_HPP
#define __FCT_HPP

class ClasseFonction : public ClasseBase {
protected:

        int             genre; // Numéro de la fonction

public:
                        ClasseFonction( int n = 0 ) { genre = n; };
                       ~ClasseFonction( ) { genre = 0; };

virtual int     isA( ) const { return _ClasseFonction; };
virtual int             isEqual( const ClasseBase& ) const;

const   int             eval ( ListeLineaire * = NULL );
};



class ClassePourTout : public ClasseBase {
protected:
        ClasseVarEntiere *v;
        ClasseExpression bi, // Valeur inférieur de la boucle
                         bs, // Valeur supérieur de la boucle
                         exp;

public:
                        ClassePourTout( ClasseVarEntiere *ve = NULL,
                                        ClasseExpression *bil = NULL,
                                        ClasseExpression *bsl = NULL,
                                        ClasseExpression *expl = NULL
```

```
)
                                    {
                                      if( ve ) v = ve;
                                      if( bil ) bi = *bil;
                                      if( bsl ) bs = *bsl;
                                      if( expl ) exp = *expl;
                                    };
                          ~ClassePourTout( );

virtual int      isA( ) const { return _ClassePourTout; };
virtual int                isEqual( const ClasseBase& ) const;

const   int                eval ( ListeLineaire * = NULL );

virtual char              *nom( ) { return "Pour tout"; };

};


class ClasseIlExiste : public ClasseBase {
protected:
        ClasseVarEntiere *v;
        ClasseExpression bi, // Valeur inférieur de la boucle
                         bs, // Valeur supérieur de la boucle
                         exp;

public:
                         ClasseIlExiste( ClasseVarEntiere *ve = NULL,
                                         ClasseExpression *bil = NULL,
                                         ClasseExpression *bsl = NULL,
                                         ClasseExpression *expl = NULL
)
                                    {
                                      if( ve ) v = ve;
                                      if( bil ) bi = *bil;
                                      if( bsl ) bs = *bsl;
                                      if( expl ) exp = *expl;
                                    };
                          ~ClasseIlExiste( );

virtual int      isA( ) const { return _ClasseIlExiste; };
virtual int                isEqual( const ClasseBase& ) const;

const   int                eval ( ListeLineaire * = NULL );
};
#endif
```

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

#include "fct.hpp"

long pile_val[MAXPILEVAL];
int  sommet_pile_val = 0;

int resultat( void ) { return (int)pile_val[ --sommet_pile_val ]; };


int NextSegment = 0; // Donne le numéro du prochain segment que l'on p
ourra
                     // définir pour les fonctions 'HEAD' et 'TAIL'



int ClasseFonction::isEqual( const ClasseBase& b ) const
{
  return ( this->isA( ) == b.isA( ) );
}



const   int ClasseFonction::eval( ListeLineaire *l )
{
  long x, y, s, j;
  int adr;
  ClasseTabSeg *t1, *t2, *t3, *t;
  ClasseSegment *ss;
  ListeLineaire l1, l2;
  char *c;
  int lg1, lg2, lg3, bi1, bi2, bi3, bs1, bs2, bs3, arret, v1, v2, i, b
i, bs;
  int b, lg;
  ClasseExpression e1, e2;

  if( ( ( genre >= _ClassePlus ) && ( genre <= _ClasseMod ) ) ||
      ( ( genre >= _ClasseAnd ) && ( genre <= _ClasseLt ) )
    )
  {
    y = pile_val[--sommet_pile_val];

    x = pile_val[--sommet_pile_val];

    switch( genre ) {
    case _ClassePlus:
        s = x + y;
        break;
    case _ClasseMoins:
        s = x - y;
        break;
    case _ClasseFois:
        s = x * y;
        break;
    case _ClasseDiv:
        s = (long)((double)x / (double)y);
        break;
    case _ClasseMod:
        s = x % y;
        break;
    case _ClasseAnd:
```

```
        s = x && y;
        break;
    case _ClasseOr:
        s = x || y;
        break;
    case _ClasseEq:
        s = x == y;
        break;
    case _ClasseNe:
        s = x != y;
        break;
    case _ClasseLt:
        s = x < y;
        break;
    case _ClasseLe:
        s = x <= y;
        break;
    case _ClasseGt:
        s = x > y;
        break;
    case _ClasseGe:
        s = x >= y;
        break;
    };

    pile_val[sommet_pile_val++] = s;
}
else if( ( genre == _ClasseNeg ) || ( genre == _ClasseNot ) )
{
    x = pile_val[--sommet_pile_val];

    switch( genre ) {
    case _ClasseNot:
        s = !x;
        break;
    case _ClasseNeg:
        s = -x;
        break;
    };

    pile_val[sommet_pile_val++] = s;
}
else if( ( genre >= _ClasseSomme ) && ( genre <= _ClassePermute ) )
{

    t = (ClasseTabSeg *)l->courant( );
    l->avancer( );
    bi = (*t).BorneInf( );
    bs = (*t).BorneSup( );
    lg = bs - bi +1;
    adr = t->adresse( );

    switch( genre ) {
    case _ClasseSomme:
        s = 0l;
        for( i = bi; i <= bs; i++ )
        {
          s += (long)( (*t)( i ) );
        };
        break;
    case _ClasseProduit:
        s = 1l;
        for( i = bi; i <= bs; i++ )
        {
```

```
        s *= (long)( (*t)( i ) );
      };
      break;
case _ClasseMin:
      s = MAXVAL + 1;
      for( i = bi; i <= bs; i++ )
      {
        j = (long)( (*t)( i ) );
        if( j < s ) s = j;
      };
      break;
case _ClasseMax:
      s = MINVAL - 1;
      for( i = bi; i <= bs; i++ )
      {
        j = (long)( (*t)( i ) );
        if( j > s ) s = j;
      };
      break;
case _ClasseLong:
      if( bi <= bs ) s = (long)( bs - bi + 1 );
      else s = 0l;
      break;
case _ClasseFirst:
      if( bi <= bs ) s = (long)((*t)(bi));
      else s = MINVAL-1;
      break;
case _ClasseLast:
      if( bi <= bs ) s = (long)((*t)(bs));
      else s = MAXVAL+1;
      break;
case _ClasseEmpty:
      s = (long)( bi > bs );
      break;
case _ClasseTriCroi:
      arret = 0;

      i = bi;

      if( lg <= 1 )
      {
        s = 1l;
      }
      else
      {
        i = bi + 1;
        v1 = (*t)(bi);
        v2 = (*t)(i);
        arret = ( v1 > v2 );
        while( !arret && (i < bs) )
        {
          i++;
          v1 = v2;
          v2 = (*t)(i);
          arret = ( v1 > v2 );
        };

        s = (long)(!arret);
      }
      break;
case _ClasseTriDec:
      arret = 0;

      i = bi;
```

```
    if( lg <= 1 )
    {
      s = 1l;
    }
    else
    {
      i = bi + 1;
      v1 = (*t)(bi);
      v2 = (*t)(i);
      arret = ( v1 < v2 );
      while( !arret && (i < bs) )
      {
        i++;
        v1 = v2;
        v2 = (*t)(i);
        arret = ( v1 < v2 );
      };

      s = (long)(!arret);
    }
    break;
case _ClasseTriStrCroi:
    arret = 0;

    i = bi;

    if( lg <= 1 )
    {
      s = 1l;
    }
    else
    {
      i = bi + 1;
      v1 = (*t)(bi);
      v2 = (*t)(i);
      arret = ( v1 >= v2 );
      while( !arret && (i < bs) )
      {
        i++;
        v1 = v2;
        v2 = (*t)(i);
        arret = ( v1 >= v2 );
      };

      s = (long)(!arret);
    }
    break;
case _ClasseTriStrDec:
    arret = 0;

    i = bi;

    if( lg <= 1 )
    {
      s = 1l;
    }
    else
    {
      i = bi + 1;
      v1 = (*t)(bi);
      v2 = (*t)(i);
      arret = ( v1 <= v2 );
      while( !arret && (i < bs) )
```

```
            {
              i++;
              v1 = v2;
              v2 = (*t)(i);
              arret = ( v1 <= v2 );
            };

            s = (long)(!arret);
          }
          break;
      case _ClassePair:
          arret = 0;

          i = bi;

          while( !arret && ( i <= bs ) )
          {
            arret = ( (*t)( i ) % 2 ) != 0;
            i++;
          };

          s = (long)(!arret); // Le résultat est vrai si 'arret' est fau
x.
          break;
      case _ClasseInchange:
          arret = 0;
          i = bi;

          while ( !arret && ( i <= bs ) )
          {
            arret = ( lire_mem( t->adresse( ) + i ) != lire_mem0( t->adr
esse( ) + i ) );
            i++;
          };

          s = (long)(!arret);
          break;
      case _ClasseExamine:
          arret = 0;
          i = bi;

          while ( !arret && ( i <= bs ) )
          {
            arret = !( exa[ adr + i ] & EXAMINE );
            i++;
          };

          s = (long)(!arret);
          break;
      case _ClassePermute:

          if( t->isA( ) == _ClasseTableau )
             t = (ClasseTableau *)t;
          else
             t = (*(ClasseSegment *)t).tableau( );

          for( i = bi; i <= bs; i++)
             if( exa[ adr + i ] & PERMUTE )
                  exa[ adr + i] -= PERMUTE;

          arret = 0;

          i = bi;
```

```
        while( !arret && (i <= bs) )
        {
          j = bi;
          b = 0;
          v1 = (*t)(i);
          while( !b && ( j <= bs ) )
          {
            v2 = lire_mem0( adr + j );
            b = ( v1 == v2 );
            if( b )
            {
              b = ( exa[ adr + j ] & PERMUTE ) == 0;
            };
            j++;
          };
          if( b ) exa[ adr + j-1 ] += PERMUTE;
          else arret = 1;
          i++;
        };

        s = (long)( i > bs );
        break;

    }; /* switch( genre ) */

    pile_val[sommet_pile_val++] = s;

  }
  else if( ( genre >= _ClasseIsSegment ) && ( genre <= _ClassePermutat
ion ) )
  {
    t1 = (ClasseTabSeg *)l->courant( );
    l->avancer( );
    t2 = (ClasseTabSeg *)l->courant( );
    l->avancer( );

    bi1 = (*t1).BorneInf( );
    bs1 = (*t1).BorneSup( );
    bi2 = (*t2).BorneInf( );
    bs2 = (*t2).BorneSup( );

    lg1 = bs1 - bi1 + 1;
    lg2 = bs2 - bi2 + 1;

    switch( genre ) {
    case _ClasseEgal:
        if( lg1 != lg2 )
        {
          s = 0l;
          break;
        };
        b = 1;

        i = bi1;
        j = bi2;

        while( b && (i <= bs1) )
        {
          b = ( (*t1)(i) == (*t2)(j) );
          i++;
          j++;
        };

        s = (long)b;
```

```
        break;
    case _ClassePermutation:
        if( lg2 != lg1 )
        {
          s = 0l;
          break;
        };

        c = (char *)calloc( lg1, sizeof( char ) );

        for( i = 0; i < lg1; i++) c[i] = 0;

        arret = 0;

        i = bi1;

        while( !arret && (i <= bs1) )
        {
          j = bi1;
          b = 0;
          while( !b && ( j <= bs2 ) )
          {
            b = ( (*t1)(i) == (*t2)(j) );
            if( b )
            {
              b = ( c[j-bi2] == 0 );
            };
            j++;
          };
          if( b ) c[j-bi2-1] = 1;
          else arret = 1;
          i++;
        };

        free( c );

        s = (long)( i > bs1 );

        break;
    case _ClasseIsSegment:
        switch( t1->isA( ) ) {
        case _ClasseTableau:
            switch( t2->isA( ) ) {
            case _ClasseTableau:
                    error( 19 ); // Un tableau ne peut pas être un sou
s ensemble
                                 // d'un autre tableau.
                    return 0;
            case _ClasseSegment:
                    b = ( strcmp( t1->nom( ), ( ((ClasseSegment *)t2)-
>tableau( ) )->nom( ) ) == 0 ) &&
                        ( bi1 <= bi2 ) && ( bi2 <= bs2 ) && ( bs2 <= b
s1 );
                    break;
            };
            break;
        case _ClasseSegment:
            switch( t2->isA( ) ) {
            case _ClasseTableau:
                    b = ( strcmp( t2->nom( ), ( ((ClasseSegment *)t1)-
>tableau( ) )->nom( ) ) == 0 ) &&
                        ( bi1 <= bi2 ) && ( bi2 <= bs2 ) && ( bs2 <= b
s1 );
```

```
                    break;
            case _ClasseSegment:
                    b = ( strcmp( ( ((ClasseSegment *)t1)->tableau( )
)->nom( ), ( ((ClasseSegment *)t2)->tableau( ) )->nom( ) ) == 0 ) &&
                        ( bi1 <= bi2 ) && ( bi2 <= bs2 ) && ( bs2 <= b
s1 );
                    break;
            };
            break;
        };

        s = (long)b;
        break;
    case _ClassePrefixe:
        switch( t1->isA( ) ) {
        case _ClasseTableau:
            switch( t2->isA( ) ) {
            case _ClasseTableau:
                    error( 19 ); // Un tableau ne peut pas être un sou
s ensemble
                                 // d'un autre tableau.
                    return 0;
            case _ClasseSegment:
                    b = ( strcmp( t1->nom( ), ( ((ClasseSegment *)t2)-
>tableau( ) )->nom( ) ) == 0 ) &&
                        ( bi1 == bi2 ) && ( bi2 <= bs2 ) && ( bs2 <= b
s1 );
                    break;
            };
            break;
        case _ClasseSegment:
            switch( t2->isA( ) ) {
            case _ClasseTableau:
                    b = ( strcmp( t2->nom( ), ( ((ClasseSegment *)t1)-
>tableau( ) )->nom( ) ) == 0 ) &&
                        ( bi1 == bi2 ) && ( bi2 <= bs2 ) && ( bs2 <= b
s1 );
                    break;
            case _ClasseSegment:
                    b = ( strcmp( ( ((ClasseSegment *)t1)->tableau( )
)->nom( ), ( ((ClasseSegment *)t2)->tableau( ) )->nom( ) ) == 0 ) &&
                        ( bi1 == bi2 ) && ( bi2 <= bs2 ) && ( bs2 <= b
s1 );
                    break;
            };
            break;
        };

        s = (long)b;
        break;
    case _ClasseSuffixe:
        switch( t1->isA( ) ) {
        case _ClasseTableau:
            switch( t2->isA( ) ) {
            case _ClasseTableau:
                    error( 19 ); // Un tableau ne peut pas être un sou
s ensemble
                                 // d'un autre tableau.
                    return 0;
            case _ClasseSegment:
                    b = ( strcmp( t1->nom( ), ( ((ClasseSegment *)t2)-
>tableau( ) )->nom( ) ) == 0 ) &&
                        ( bi1 <= bi2 ) && ( bi2 <= bs2 ) && ( bs2 == b
s1 );
```

```
                    break;
                };
                break;
            case _ClasseSegment:
                switch( t2->isA( ) ) {
                case _ClasseTableau:
                    b = ( strcmp( t2->nom( ), ( ((ClasseSegment *)t1)-
>tableau( ) )->nom( ) ) == 0 ) &&
                        ( bi1 <= bi2 ) && ( bi2 <= bs2 ) && ( bs2 == b
s1 );
                    break;
                case _ClasseSegment:
                    b = ( strcmp( ( ((ClasseSegment *)t1)->tableau( )
)->nom( ), ( ((ClasseSegment *)t2)->tableau( ) )->nom( ) ) == 0 ) &&
                        ( bi1 <= bi2 ) && ( bi2 <= bs2 ) && ( bs2 == b
s1 );
                    break;
                };
                break;
            };
            s = (long) b;
            break;


    }; /* switch( genre ) */

    pile_val[sommet_pile_val++] = s;

  }
  else if( genre == _ClasseConcat )
  {
    t1 = (ClasseTabSeg *)l->courant( );
    l->avancer( );
    t2 = (ClasseTabSeg *)l->courant( );
    l->avancer( );
    t3 = (ClasseTabSeg *)l->courant( );
    l->avancer( );

    bi1 = (*t1).BorneInf( );
    bs1 = (*t1).BorneSup( );

    bi2 = (*t2).BorneInf( );
    bs2 = (*t2).BorneSup( );

    bi3 = (*t3).BorneInf( );
    bs3 = (*t3).BorneSup( );

    lg1 = bs1 - bi1 + 1;
    lg2 = bs2 - bi2 + 1;
    lg3 = bs3 - bi3 + 1;

    b = (lg3 == (lg1 + lg2));

    i = bi1;
    j = bi3;

    while( b && (i <= bs1) )
    {
      b = ( (*t1)(i) == (*t3)(j) );
      i++;
      j++;
    };

    i = bi2;
```

```
    while( b && (i <= bs2) )
    {
      b = ( (*t2)(i) == (*t3)(j) );
      i++;
      j++;
    };

    s = (long)b;

    pile_val[sommet_pile_val++] = s;
  }
  else if( ( genre == _ClasseHead ) || ( genre == _ClasseTail ) )
  {
    t1 = (ClasseTabSeg *)l->courant( );
    l->avancer( );
    ss = (ClasseSegment *)l->courant( );
    l->avancer( );

    bi1 = (*t1).BorneInf( );
    bs1 = (*t1).BorneSup( );

    lg1 = bs1 - bi1 + 1;

    if( t1->isA( ) == _ClasseTableau ) t = (ClasseTableau *)t1;
    else t = (*(ClasseSegment *)t1).tableau( );

    l1.creer( );
    l2.creer( );
    ((ss->bi).Liste( ) )->detruire( );
    ((ss->bs).Liste() )->detruire( );

    switch( genre ) {
    case _ClasseHead:
        l1.inserer( new ClasseConstante( bi1 ) );
        e1 = ClasseExpression( &l1 );
        l2.inserer( new ClasseConstante( bs1 - 1 ) );
        e2 = ClasseExpression( &l2 );
        *ss = ClasseSegment( (ClasseTableau *)t, ss->nom( ), &e1, &e2
);
        break;
    case _ClasseTail:
        l1.inserer( new ClasseConstante( bi1 + 1 ) );
        e1 = ClasseExpression( &l1 );
        l2.inserer( new ClasseConstante( bs1 ) );
        e2 = ClasseExpression( &l2 );
        *ss = ClasseSegment( (ClasseTableau *)t, ss->nom( ), &e1, &e2
);
        break;
    }; /* switch( genre ) */

  };

  return 0;
}



int     ClassePourTout::isEqual( const ClasseBase& b ) const
{
  ClassePourTout *p;
  int vbi, vbs;

  if( this->isA( ) != b.isA( ) ) return 0;
```

```
  p = &((ClassePourTout&)b);
  return ( this == p );
}


const   int     ClassePourTout::eval( ListeLineaire *l )
{
  int i, b, vbi, vbs;

  bi.eval( );
  vbi = (int)pile_val[--sommet_pile_val];

  bs.eval( );
  vbs = (int)pile_val[--sommet_pile_val];

  b = 1;
  i = vbi;

  while( b && ( i <= vbs ) )
  {
    (*v) = i;
    exp.eval( );
    b = (int)pile_val[ --sommet_pile_val ];
    i++;
  };

  pile_val[ sommet_pile_val++ ] = (long)b;

  return 0;
}


int     ClasseIlExiste::isEqual( const ClasseBase& b ) const
{
  ClasseIlExiste *p;

  if( this->isA( ) != b.isA( ) ) return 0;
  p = &((ClasseIlExiste&)b);
  return ( this == p );
}

const   int     ClasseIlExiste::eval( ListeLineaire *l )
{
  int i, b, vbi, vbs;

  bi.eval( );
  vbi = pile_val[ --sommet_pile_val ];

  bs.eval( );
  vbs = pile_val[ --sommet_pile_val ];

  b = 0;
  i = vbi;

  while( !b && ( i <= vbs ) )
  {
    (*v) = i;
    exp.eval( );
    b = (int)pile_val[ --sommet_pile_val ];
    i++;
  };

  pile_val[ sommet_pile_val++ ] = (long)b;
```

```
  return 0;
}
```

Fichiers INST.HPP et INST.CPP

Fichiers des instructions du langage

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "cbase.hpp"
#include "expr.hpp"


extern int running;



int ajo_adr( int eti );

int adr_eti( int eti );

int ajo_inst_si( ClasseExpression *b, unsigned adr );

int ajo_inst_goto( unsigned adr );

int ajo_inst_eval( ClasseBase *o );

int ajo_inst_let( ClasseBase *v, ClasseExpression *e1, ClasseExpressio
n *e2 );

int ajo_inst_stop( void );

int ajo_inst_pri( const char *p );

int ajo_inst_priv( ClasseBase *v, ClasseExpression *e1 );

int ajo_inst_read( ClasseBase *v, ClasseExpression *e1 );

int ajo_inst_flush( void );

int ajo_inst_nl( );

void run_program( int eti );

void step_program( int eti );
```

```
#include <stdio.h>
#include <string.h>

#include "inst.hpp"
#include "var.hpp"


#define SI                1
#define GOTO              2
#define EVAL              3
#define LET               4
#define STOP              5
#define PRINT             6
#define PRINTVAR          7
#define PRINTNL           8
#define READ              9
#define FLUSH            10

typedef struct etiquette {
        unsigned num;
        unsigned adr;
} etiquette;

typedef struct ins_si {
        ClasseExpression *b;
        unsigned adr;
} ins_si;

typedef struct ins_goto {
        unsigned adr;
} ins_goto;

typedef struct ins_eval {
        ClasseBase *o;
} ins_eval;

typedef struct ins_let {
        ClasseBase *v;
        ClasseExpression *e1, *e2;
} ins_let;

typedef struct ins_pri {
        char *p;
} ins_pri;

typedef struct ins_priv {
        ClasseBase *v;
        ClasseExpression *e1;
} ins_priv;


typedef union instruction {
        ins_si si;
        ins_goto go;
        ins_eval eval;
        ins_let let;
        ins_pri pri;
        ins_priv priv;
} instruction;

typedef struct prog {
        int genre;
        instruction i;
} prog;
```

```
#define MAXINSTRUCTION 1000

prog program[MAXINSTRUCTION];
unsigned lastins = 0;


#define MAXLABEL 100
etiquette label[MAXLABEL];
unsigned lastlab = 0;


unsigned cour_inst = 0;
int running = 0;



int ajo_adr( int eti  )
{
  if( lastlab == MAXLABEL ) return 0;
  label[ lastlab ].num = eti;
  label[ lastlab ].adr = lastins;
  lastlab++;
  return 1;
}


int adr_eti( int eti )
{
  int i;

  i = 0;
  while( i < lastlab )
  {
    if( eti == label[ i ].num ) return label[ i ].adr;
    i++;
  };
  return -1;
}

void eval_inst( prog inst )
{
  int g, j, v;
  ClasseBase *o;

  switch( inst.genre ) {
  case SI:
        (inst.i.si.b)->eval( );
        if( resultat( ) )
        {
          cour_inst = adr_eti( inst.i.si.adr );
          return ;
        };
        break;
  case GOTO:
        cour_inst = adr_eti( inst.i.go.adr );
        return;
        break;
  case EVAL:
        printf( "Resultat : %s = ", inst.i.eval.o->nom( ) );
        inst.i.eval.o->eval( );
        printf( "%d\n", resultat( ) );
```

```
        break;
case LET:
        o = inst.i.let.v;
        g = o->isA( );
        if( inst.i.let.e1 )
        {
          inst.i.let.e1->eval( );
          j = resultat( );
        };
        if( inst.i.let.e2 )
        {
          inst.i.let.e2->eval( );
          v = resultat( );
        };
        switch( g ) {
        case _ClasseVarEntiere:
                (*(ClasseVarEntiere *)o) = v;
                break;
        case _ClasseVarLogique:
                *((ClasseVarLogique *)o) = v;
                break;
        case _ClasseTableau:
                (*(ClasseTableau *)o)(j) = v;
                break;
        case _ClasseSegment:
                (*(ClasseSegment *)o)(j) = v;
                break;
        };
        break;
case STOP:
        running = 0;
        break;
case PRINT:
        if( inst.i.pri.p ) printf( "%s", inst.i.pri.p );
        break;
case PRINTVAR:
        o = inst.i.priv.v;
        g = o->isA( );
        if( inst.i.priv.e1 )
        {
          inst.i.priv.e1->eval( );
          j = resultat( );
        };
        switch( g ) {
        case _ClasseVarEntiere:
                o->eval( );
                printf( "%d", resultat( ) );
                break;
        case _ClasseVarLogique:
                o->eval( );
                printf( "%d", resultat( ) );
                break;
        case _ClasseTableau:
                printf( "%d", (*(ClasseTableau *)o)(j) );
                break;
        case _ClasseSegment:
                printf( "%d", (*(ClasseSegment *)o)(j) );
                break;
        };
        break;
case PRINTNL:
        printf( "\n" );
        break;
case READ:
```

```
        o = inst.i.priv.v;
        g = o->isA( );
        if( inst.i.priv.e1 )
        {
          inst.i.priv.e1->eval( );
          j = resultat( );
        };
        scanf( "%d", &v );
        switch( g ) {
        case _ClasseVarEntiere:
                (*(ClasseVarEntiere *)o) = v;
                break;
        case _ClasseVarLogique:
                *((ClasseVarLogique *)o) = v;
                break;
        case _ClasseTableau:
                (*(ClasseTableau *)o)(j) = v;
                break;
        case _ClasseSegment:
                (*(ClasseSegment *)o)(j) = v;
                break;
        };
        break;
  case FLUSH:
        fflush( stdin );
        break;
  };
  cour_inst++;
}

int ajo_inst_si( ClasseExpression *b, unsigned adr )
{
  if( lastins == MAXINSTRUCTION ) return 0;

  program[ lastins ].genre = SI;
  program[ lastins ].i.si.b = b;
  program[ lastins ].i.si.adr = adr;

  lastins++;
  return 1;
}

int ajo_inst_goto( unsigned adr )
{
  if( lastins == MAXINSTRUCTION ) return 0;

  program[ lastins ].genre = GOTO;
  program[ lastins ].i.go.adr = adr;

  lastins++;
  return 1;
}

int ajo_inst_eval( ClasseBase *o )
{
  if( lastins == MAXINSTRUCTION ) return 0;

  program[ lastins ].genre = EVAL;
  program[ lastins ].i.eval.o = o;

  lastins++;
  return 1;
}
```

```
int ajo_inst_let( ClasseBase *v, ClasseExpression *e1, ClasseExpressio
n *e2 )
{
  if( lastins == MAXINSTRUCTION ) return 0;

  program[ lastins ].genre = LET;
  program[ lastins ].i.let.v = v;
  program[ lastins ].i.let.e1 = e1;
  program[ lastins ].i.let.e2 = e2;

  lastins++;
  return 1;
}

int ajo_inst_stop( void )
{
  if( lastins == MAXINSTRUCTION ) return 0;

  program[ lastins ].genre = STOP;

  lastins++;
  return 1;
}

int ajo_inst_pri( const char *p )
{
  char *q;

  if( lastins == MAXINSTRUCTION ) return 0;

  if( p )
  {
    q = (char *)malloc( strlen( p ) );
    if( q ) strcpy( q, p );
  }
  else q = NULL;

  program[ lastins ].genre = PRINT;
  program[ lastins ].i.pri.p = q;

  lastins++;
  return 1;
}

int ajo_inst_priv( ClasseBase *v, ClasseExpression *e1 )
{
  char *q;

  if( lastins == MAXINSTRUCTION ) return 0;

  program[ lastins ].genre = PRINTVAR;
  program[ lastins ].i.priv.v = v;
  program[ lastins ].i.priv.e1 = e1;

  lastins++;
  return 1;
}

int ajo_inst_read( ClasseBase *v, ClasseExpression *e1 )
{
  if( lastins == MAXINSTRUCTION ) return 0;

  program[ lastins ].genre = READ;
  program[ lastins ].i.priv.v = v;
```

```
  program[ lastins ].i.priv.e1 = e1;

  lastins++;
  return 1;
}

int ajo_inst_flush( void )
{
  if( lastins == MAXINSTRUCTION ) return 0;

  program[ lastins ].genre = FLUSH;

  lastins++;
  return 1;
}



int ajo_inst_nl( )
{
  if( lastins == MAXINSTRUCTION ) return 0;

  program[ lastins ].genre = PRINTNL;

  lastins++;
  return 1;
}

void run_program( int eti )
{
  running = 1;
  if( eti == -1 ) cour_inst = 0;
  else cour_inst = adr_eti( eti );
  while( running )
  {
    eval_inst( program[ cour_inst ] );
    if( cour_inst >= lastins ) running = 0;
  };
}
```

Fichiers LISTELIN.HPP et LISTELIN.CPP

Fichiers des listes linéaires

```
#ifndef __CCONST_HPP
#include "cconst.hpp"
#endif

#ifndef __CBASE_HPP
#include "cbase.hpp"
#endif

#ifndef __LISTELIN_HPP
#define __LISTELIN_HPP

typedef struct typeartlistelin {
        struct typeartlistelin *suiv;
        ClasseBase *ref;
} TYPEARTLISTELIN;

class ListeLineaire : public ClasseBase {
protected:
        long n;
        struct typeartlistelin *e, *c;
public:
        ListeLineaire ( );
        ~ListeLineaire ( );


virtual int     isA( ) const { return _ClasseListeLin; };
virtual int             isEqual( const ClasseBase& ) const;


virtual const int       eval( ListeLineaire * = NULL )
                                { return ERROR; };

        int eof( );
        void creer( );
        int avancer( );
        int inserer( ClasseBase *V );
        ClasseBase *courant( );
        long nbr( ) const { return n; };
        void repositionner( );
        int supprimer( int all = 0 );
        void detruire( int all = 0 );

        ListeLineaire& operator=( ListeLineaire& L );
};

#endif
```

```
#include <alloc.h>

#include "listelin.hpp"

void ListeLineaire::creer( )
{
  n = 0l;
  e = (TYPEARTLISTELIN *)malloc(sizeof(TYPEARTLISTELIN));
  if ( e == NULL) return;
  c = e;
  c->suiv = NULL;
}

int ListeLineaire::eof( )
{ return (c->suiv == NULL);
}

int ListeLineaire::avancer( )
{ if (this->eof( )) return 0;
  c = c->suiv;
  return 1;
}

int ListeLineaire::inserer( ClasseBase *V )
{ struct typeartlistelin *p2;

  p2 = (TYPEARTLISTELIN *)malloc(sizeof(TYPEARTLISTELIN));
  if (p2 == NULL) return 0;
  p2->ref = V;
  p2->suiv = c->suiv;
  c->suiv = p2;
  n++;
  return 1;
}

ClasseBase *ListeLineaire::courant(  )
{ if (this->eof( )) return NULL;
  return (c->suiv)->ref;
}

void ListeLineaire::repositionner( )
{ c = e;
}

int ListeLineaire::supprimer( int all )
{ TYPEARTLISTELIN *p2;

  if (this->eof( )) return 0;
  p2 = c->suiv;
  c->suiv = p2->suiv;

  if (all) delete p2->ref; /* suppretion de l'article que l'on a stock
é */
  free (p2); /* On supprime la cellule de la liste chaînée */

  n--;

  return 1;
}

void ListeLineaire::detruire( int all )
{ this->repositionner( );
  while (!this->eof( )) this->supprimer( all );
}
```

```
ListeLineaire::ListeLineaire( )
{
  this->creer( );
}

ListeLineaire::~ListeLineaire( )
{
  if( this )
  { this->detruire( 0 );
    free( e );
    e = c = NULL;
  };
}

ListeLineaire& ListeLineaire::operator=( ListeLineaire& L )
{ ClasseBase *p;
  struct typeartlistelin *q;
  char oldf;

  q = L.c;
  this->detruire( );
  L.repositionner( );
  while ( !L.eof( ) )
  {
    p = L.courant( );
    this->inserer(p);
    this->avancer( );
    L.avancer( );
  };
  L.c = q;
  this->repositionner( );
  return *this;
}


int ListeLineaire::isEqual( const ClasseBase& b ) const
{
  int v;
  ListeLineaire *l2;
  ClasseBase *b1, *b2;

  if( this->isA( ) != b.isA( ) ) return 0;
  l2 = &((ListeLineaire&)b);
  if( this->nbr( ) != l2->nbr( ) ) return 0;

  if( e == l2->e ) return 1; // Si l'entête des deux listes est à la m
ême
                             // adresse, on vérifie l'égalité entre X
et X

  this->repositionner( );
  l2->repositionner( );
  v = 1;
  while( !l2->eof( ) && v )
  {
    b1 = this->courant( );
    b2 = l2->courant( );
    v = b1->isEqual( *b2 );
    this->avancer( );
    l2->avancer( );
  };
  return v;
}
```

Fichiers VAR.HPP et VAR.CPP

Fichiers des variables et des tableaux

```
#include <mem.h>
#include <string.h>

#ifndef __IOSTREAM_H
#include <iostream.h>
#endif


#include "cconst.hpp"


#include "cbase.hpp"


#include "ListeLin.hpp"

#include "expr.hpp"


#ifndef __VAR_HPP
#define __VAR_HPP


class ClasseVariable : public ClasseBase {
protected:
        char n[MAXVARLEN + 1];
        int adr;

public:
                         ClasseVariable( const char *p = NULL );
                        ~ClasseVariable( );

virtual int      isA( ) const { return _ClasseVariable; };
virtual int              isEqual( const ClasseBase& ) const;

const   int              eval ( ListeLineaire * = NULL );

virtual char            *nom( ) { return n; };
        void             nom( char *p ) { strcpy( n, p ); };

        void             adresse( int _adr );
        int              adresse( void ) const;

        int              valeur( ) {return lire_mem( adr );};

};

class ClasseVarEntiere : public ClasseVariable {
protected:

public:
                         ClasseVarEntiere( const char *p = NULL ) : Cla
sseVariable( p )
                                 { /* nom( p ); */ };
                        ~ClasseVarEntiere( ){};

virtual int      isA( ) const { return _ClasseVarEntiere; };

        int              operator=( const ClasseVarEntiere& t );
        int              operator=( int i );

const   int              eval ( ListeLineaire * = NULL );
        int              valeur( );
```

```
};

class ClasseVarLogique : public ClasseVariable {
protected:

public:
                        ClasseVarLogique( const char *p = NULL ) : Cla
sseVariable( p ) { };
                       ~ClasseVarLogique( );

virtual int     isA( ) const { return _ClasseVarLogique; };

        int             operator=( const ClasseVarLogique& t );
        int             operator=( int i );

const   int             eval ( ListeLineaire * = NULL );
        int             valeur( );

};
#endif


#ifndef __CONST
#define __CONST

class ClasseConstante : public ClasseBase {
protected:
        int v;

public:
                        ClasseConstante( int val = 0 ) { v = val; };
                       ~ClasseConstante( ) {};

virtual int     isA( ) const { return _ClasseConstante; };
virtual int             isEqual( const ClasseBase& ) const;

const   int             eval ( ListeLineaire * = NULL );

        int             operator=( int val );
        int             operator=( const ClasseConstante& c );

};

#endif


#ifndef __CTAB_HPP

#define __CTAB_HPP

class ClasseTabSeg : public ClasseBase {
protected:
        char n[MAXVARLEN + 1];
public:
                        ClasseTabSeg( const char *q = NULL )
                        { if( q ) strcpy( n, q );
                          else strcpy( n, "" );
                        };
                       ~ClasseTabSeg( ) {};

virtual int             isA( ) const { return _ClasseTabSeg; };
virtual int             isEqual( const ClasseBase& ) const{ return 0;
};
```

```
        ClasseTabSeg&   operator=( const ClasseTabSeg& t );

const   int             eval( ListeLineaire * = NULL ) { return ERROR;
};

virtual int             adresse( ) const = 0;

virtual char           *nom( ) {return n; };

/*
 * Les operateurs suivants permettent un accès à un élément du tableau
.
 * Les parenthèses sont utilisées plutôt que les crochets.
 */
        int&            operator() ( int i );
        int&            operator() ( ClasseVariable& i );
        int             operator() ( int i ) const;
        int             operator() ( ClasseVariable& i ) const;

virtual int             BorneInf( ) const = 0;
virtual int             BorneSup( ) const = 0;

};


class ClasseTableau : public ClasseTabSeg {
protected:
        char n[MAXVARLEN + 1];
        int adr, bi, bs;
        /*
           Adresse est la position du tableau dans la mémoire. 'bi' es
t la
           valeur de la borne inférieur, 'bs' la valeur de la borne su
périeur
         */

public:
                        ClasseTableau( const char *q = NULL ) : Classe
TabSeg( q ) {};
                       ~ClasseTableau( );

virtual int     isA( ) const { return _ClasseTableau; };
virtual int             isEqual( const ClasseBase& ) const;

        ClasseTableau&  operator=( const ClasseTableau& t );

        void            adresse( int _adr, int _bi, int _bs );
        int             adresse( ) const { return adr; };


        int             BorneInf( ) const;
        int             BorneSup( ) const;
};

/*
 Cette classe va être utilisée lors de l'évaluation des expressions da
ns
 lesquelles interviennent les variables tableaux. Une instance de cett
e
 classe représente une variable simple d'un tableau, c'est-à-dire la
 i ème variable du tableau. L'indice peut être obtenu par une valeur e
ntière
 ou par une variable de type entière.
```

```
 */

class ClasseVarInd : public ClasseBase {
protected:
        ClasseTabSeg *tab;

public:
                        ClasseVarInd( ClasseTabSeg *t = NULL );
                       ~ClasseVarInd( );

virtual int     isA( ) const { return _ClasseVarInd; };
virtual int             isEqual( const ClasseBase& ) const;

const   int             eval( ListeLineaire * = NULL );

};


/*
 Un segment est une partie d'un tableau. Pour cela, on associe à chaqu
e
 segment le tableau auquel il se rapporte. Un segment peut avoir des
 bornes fixes et/ou variables.
 */


class ClasseSegment : public ClasseTabSeg {
protected:
        ClasseTableau *tab;
public:
        ClasseExpression bi, bs;

                        ClasseSegment( ClasseTableau *t = NULL,
                                       const char *p = NULL,
                                       ClasseExpression *_bi = NULL,
                                       ClasseExpression *_bs = NULL );
                       ~ClasseSegment( );

virtual int     isA( ) const { return _ClasseSegment; };
virtual int             isEqual( const ClasseBase& ) const;

        ClasseSegment&  operator=( const ClasseSegment& t );

        int             BorneInf( ) const;
        int             BorneSup( ) const;

        ClasseTableau  *tableau( ) const { return tab; };

virtual int             adresse( ) const { return tab->adresse( ); };
};
#endif
```

```
#include <stdlib.h>
#include <string.h>

#include "var.hpp"


ListeLineaire env;

int *mem, *mem0, *exa;
int nadr;

#define MAXMEM 1000

void init_mem( )
{
  int i;

  mem = (int *)calloc(MAXMEM, sizeof(int));
  if (mem)
  { nadr = 0;
    for( i = 0; i < MAXMEM; i++ ) mem[ i ] = ENT_NON_DEF;
  }
  else
  {
    /* Impossible d'allouer de la place pour la mémoire */
    exit(99);
  };
}

void term_mem( )
{
  if (mem) free( mem );
  nadr = 0;
}

static ClasseVarEntiere *ve;
static ClasseVarLogique *vl;

ClasseVarEntiere *allouer_var_ent( const char *nom )
{

  ve = new ClasseVarEntiere( nom );

  if ( nadr >= MAXMEM ) return NULL;

  ve->adresse( nadr );

  env.repositionner( );
  if( !env.inserer( ve ) ) return NULL;
  *(mem + nadr) = ENT_NON_DEF;
  nadr ++;
  return ve;
}

ClasseVarLogique *allouer_var_log( const char *nom )
{

  vl = new ClasseVarLogique( nom );

  if ( nadr >= MAXMEM ) return NULL;

  vl->adresse( nadr );
```

```
  env.repositionner( );
  if( !env.inserer( vl ) ) return NULL;
  *(mem + nadr) = ENT_NON_DEF;
  nadr ++;
  return vl;
}

ClasseTableau *tt;

ClasseTableau *allouer_tab( const char *nom, int bi, int bs )
{
  int i;

  tt = new ClasseTableau( nom );

  if ( (nadr + bs - bi + 1) >= MAXMEM ) return NULL;

  tt->adresse( nadr - bi, bi, bs );

  env.repositionner( );
  if ( !env.inserer( tt ) ) return NULL;
  for( i = bi; i <= bs; i++ ) *(mem + nadr + i - bi) = ENT_NON_DEF;
  nadr = nadr + bs - bi + 1;
  return tt;
}


static ClasseSegment *FF;

ClasseSegment *allouer_seg( ClasseTableau *t, const char *nom,
                            ClasseExpression *_bi, ClasseExpression *_
bs )
{
  FF = new ClasseSegment( t, nom, _bi, _bs );

  if( env.inserer( FF ) ) return FF;
  else return NULL;
}


int lire_mem( int adr )
{
  if (( adr >= 0) && (adr < nadr))
  { if( !(exa[ adr ] & EXAMINE) ) exa[ adr ] += EXAMINE;
    return mem[ adr ];
  }
  else return 0;
}

void ecrire_mem( int adr, int val )
{
  if (( adr >= 0) && (adr < nadr))
  { mem[ adr ] = val;
  }
}




int lire_mem0( int adr )
{
  if (( adr >= 0) && (adr < nadr))
  { return mem0[ adr ];
```

```
  }
  else return 0;
}

void init_mem0( )
{ int i;

  mem0 = (int *)calloc(nadr, sizeof(int));
  if (mem0)
  {
    for( i = 0; i < nadr; i++) *(mem0 + i ) = *(mem + i );
  }
  else
  {
    /* Impossible d'allouer de la place pour la mémoire */
    exit(99);
  };

  exa = (int *)calloc(nadr, sizeof(int));
  if (exa)
  {
    for( i = 0; i < nadr; i++) exa[i] = 0;
  }
  else
  {
    /* Impossible d'allouer de la place pour la mémoire */
    exa = NULL;
    exit(99);
  };
}

void term_mem0( )
{
  if( mem0 ) free( mem0 );
  if( exa ) free( exa );
}


ClasseBase *existe_dans_env( ClasseBase *o)
{
  ClasseBase *c;
  int b;

  env.repositionner( );
  c = NULL;
  b = 0;
  while( !b && !env.eof( ) )
  {
    c = env.courant( );
    b = o->isEqual( *c );
    env.avancer( );
  };

  if( b ) return c;
  else return NULL;
}


ClasseBase *dans_env( const char *p )
{
  ClasseBase *c;
  int b;

  env.repositionner( );
```

```
  c = NULL;
  b = 0;
  while( !b && !env.eof( ) )
  {
    c = env.courant( );
    b = !strcmp( p, c->nom( ) );
    env.avancer( );
  };

  if( b ) return c;
  else return NULL;
}


ClasseVariable::ClasseVariable( const char *p )
{
  if ( p )
  {
    strcpy( n, p );
    adr = -1;
  }
  else
  {
    strcpy( n, "" );
    adr = -1;
  };
}

ClasseVariable::~ClasseVariable( )
{
  strcpy ( n, "" );
  adr = -1;
}

int ClasseVariable::isEqual( const ClasseBase& b ) const
{
  ClasseVariable *v;

  return ( !( strcmp( n, ( ( const ClasseVariable& )b).nom() ) ) );
}

int ClasseVariable::adresse( void ) const
{
  ClasseBase *o;
  int b;

  return adr;
}

void ClasseVariable::adresse( int _adr )
{
/*  ClasseBase *o;
  int b;
  ListeLineaire l;

  l = env;
  l.repositionner( );
  b = 0;
  while( !b && !l.eof( ) )
  {
    o = env.courant( );
    b = o->isEqual( *this );
    if( b ) b = ( o->isA( ) == _ClasseVariable );
```

```
    if( b ) b = ( o->isA( ) == _ClasseVarEntiere );
    if( b ) b = ( o->isA( ) == _ClasseVarLogique );
    l.avancer( );
  };

  if( b ) ((ClasseVariable *)o)->adr = _adr; */
  adr = _adr;
}


const int ClasseVariable::eval( ListeLineaire *l )
{
  return ERROR;
}

int ClasseVarEntiere::valeur( )
{
  int j, a;

  j = lire_mem( adr );
  return j;
}


int ClasseVarEntiere::operator=( const ClasseVarEntiere& t )
{
  int j;

  j = lire_mem( t.adresse( ) );
  ecrire_mem( this->adresse( ), j );
  return j;
}

int ClasseVarEntiere::operator=( int i )
{  ecrire_mem( this->adresse( ), i );
  return i;
}

const   int ClasseVarEntiere::eval ( ListeLineaire *l )
{
  int j;

  j = lire_mem( this->adresse( ) );

  if( sommet_pile_val == MAXPILEVAL )
  {
    return ERROR;
  };
  pile_val[ sommet_pile_val++ ] = (long)j;
  return 0;
}



int ClasseVarLogique::valeur( )
{
  int j;

  j = lire_mem( this->adresse( ) );
  return j;
}

int ClasseVarLogique::operator=( const ClasseVarLogique& t )
{
```

```
  int j;

  j = lire_mem( t.adresse( ) );
  ecrire_mem( this->adresse( ), ( j ? 1 : 0 ) );
  return ( j ? 1 : 0 );
}

int ClasseVarLogique::operator=( int i )
{
  ecrire_mem( this->adresse( ), ( i ? 1 : 0 ) );
  return ( i ? 1 : 0 );
}


const   int ClasseVarLogique::eval ( ListeLineaire *l )
{
  int j;

  j = lire_mem( this->adresse( ) );

  if( j ) j = 1;
  else j = 0;

  if( sommet_pile_val == MAXPILEVAL )
  {
    return ERROR;
  };
  pile_val[ sommet_pile_val++ ] = (long)j;
  return 0;
}


int ClasseConstante::isEqual( const ClasseBase& b ) const
{
  return ( v == ((ClasseConstante&)b).v ) ;
}



const   int ClasseConstante::eval ( ListeLineaire *l )
{
  if( sommet_pile_val == MAXPILEVAL )
  {
    return ERROR;
  };
  pile_val[ sommet_pile_val++ ] = (long)v;
  return 0;
}

int ClasseConstante::operator=( int val )
{
  v = val;
  return v;
}

int ClasseConstante::operator=( const ClasseConstante& c )
{
  v = c.v;
  return v;
}



int VNULL = 0;
```

```
int& ClasseTabSeg::operator() ( int i )
{
  int adr;

  adr = adresse( );
  if (( i < BorneInf( ) ) || ( BorneSup( ) < i ))
  {
    return VNULL;
  };

  exa[ adr + i ] |= EXAMINE;
  return mem[ adr + i ];
}


int& ClasseTabSeg::operator() ( ClasseVariable& v )
{
  int i;
  int adr;

  adr = adresse( );

  i = v.valeur( );

  if (( i < BorneInf( ) ) || ( BorneSup( ) < i ))
  {
    return VNULL;
  };

  exa[ adr + i ] |= EXAMINE;

  return mem[ adr + i ];
}

int ClasseTabSeg::operator() ( int i ) const
{
  int adr;

  adr = adresse( );
  if (( i < BorneInf( ) ) || ( BorneSup( ) < i ))
  {
    return VNULL;
  };

  exa[ adr + i ] |= EXAMINE;

  return mem[ adr + i ];
}


int ClasseTabSeg::operator() ( ClasseVariable& v ) const
{
  int i;
  int adr;

  adr = adresse( );
  i = v.valeur( );

  if (( i < BorneInf( ) ) || ( BorneSup( ) < i ))
  {
    return VNULL;
  };
```

```
  exa[ adr + i ] |= EXAMINE;

  return mem[ adr + i ];
}



ClasseTableau::~ClasseTableau( )
{
  strcpy( n, "" );
  adr = -1;
}

int ClasseTableau::isEqual( const ClasseBase& b) const
{
  return ( !( strcmp( n, ((const ClasseTableau&)b).nom() ) ) );
}


ClasseTableau&  ClasseTableau::operator=( const ClasseTableau& t )
{
  int j;
  int bs, bi, bs1, bi1, a, a1;

  bs = this->BorneSup( );
  bi = this->BorneInf( );
  bs1 = t.BorneSup( );
  bi1 = t.BorneInf( );

  a = this->adresse( );
  a1 = t.adresse( );

  if ((bs - bi) != (bs1 - bi1))
  {
    return *this;
  };
  for ( j = bi; j <= bs; j++)
    ecrire_mem( a + j, lire_mem( a1 + bi1 - (j - bi) ) );
  return *this;
}



void ClasseTableau::adresse( int _adr, int _bi, int _bs )
{
  adr = _adr;
  bi = _bi;
  bs = _bs;
}

int ClasseTableau::BorneInf( ) const
{ return bi;
}

int ClasseTableau::BorneSup( ) const
{ return bs;
}



ClasseVarInd::ClasseVarInd( ClasseTabSeg *t )
{
```

```
  tab = t;
}

ClasseVarInd::~ClasseVarInd( )
{
  tab = NULL;
}

int             ClasseVarInd::isEqual( const ClasseBase& c ) const
{
  int b;


  b = ( tab == ((const ClasseVarInd&)c).tab);
  return b;
}



const   int ClasseVarInd::eval( ListeLineaire *l )
{
  int i, j;


  if( sommet_pile_val == 0 )
  {
    return ERROR;
  };

  i = (int)pile_val[ --sommet_pile_val ];
  j = (*tab)( i );

  pile_val[ sommet_pile_val++ ] = (long)j;
  return 0;
}



ClasseSegment::ClasseSegment( ClasseTableau *t, const char *p,
                              ClasseExpression *_bi,
                              ClasseExpression *_bs ) : ClasseTabSeg(
p )
{
  tab = t;
  if( _bi ) bi = *_bi;
  if( _bs ) bs = *_bs;
}

ClasseSegment::~ClasseSegment( )
{
  strcpy( n, "" );
  tab = NULL;
  (bi.Liste( ) ) -> detruire( );
  (bs.Liste())->detruire();
}

int ClasseSegment::isEqual( const ClasseBase& c) const
{
  return ( strcmp( n, ((const ClasseSegment&)c).n ) == 0 );
}


ClasseSegment& ClasseSegment::operator=( const ClasseSegment& t )
{
```

```
  strcpy( n, t.n );
  tab = t.tab;
  bi =t.bi;
  bs = t.bs;

  return *this;
}


int ClasseSegment::BorneInf( ) const
{ bi.eval( );
  return resultat( );
}

int ClasseSegment::BorneSup( ) const
{
  bs.eval( );
  return resultat( );
}
```

Fichier PRINCIPA.HPP

Fichier de traduction

```
#ifndef __PRINCIPAL__

 jmp_buf jumper;

#else

 extern jmp_buf jumper;

#endif

#ifndef __PRINCIPAL__
#define MAXBUF 2500

char buf[ MAXBUF+1];
int  lenbuf;
FILE *in;

ClasseVarEntiere *ve;
ClasseVarLogique *vl;
ClasseVarInd     *vi;
ClasseConstante  *cc;

ClasseTableau    *t;
ClasseSegment    *FF;

ClasseBase       *o, *o1, *o2, *o3;
ClasseExpression *e1, *e2, *e3;
ListeLineaire    *l;
ClasseFonction   *f;


#else
#define MAXBUF 1000

extern char buf[ MAXBUF + 1 ];
extern int  lenbuf;
extern FILE *in;

extern ClasseVarEntiere *ve;
extern ClasseVarLogique *vl;
extern ClasseVarInd      *vi;
extern ClasseConstante  *cc;

extern ClasseTableau    *t;
extern ClasseSegment  *FF;

extern ClasseBase       *o, *o1, *o2, *o3;
extern ClasseExpression *e1, *e2, *e3;
extern ListeLineaire    *l;
extern ClasseFonction   *f;


#endif

void read_buf( char *buf, unsigned max );
char *skip_white( char *s );

void error( ListeLineaire *trad, char *s, ... );

void warning( char *s, ... );

char *whites( char *p );
```

```
char *traduction( ListeLineaire *trad, char *s, char term );
```

Fichier PRINCIPA.CPP

Fichier de traduction

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdarg.h>
#include <setjmp.h>
#include <stdlib.h>

#include "cconst.hpp"
#include "cbase.hpp"
#include "listelin.hpp"
#include "var.hpp"
#include "fct.hpp"
#include "expr.hpp"

#include "inst.hpp"

#define __PRINCIPAL__

#include "principal.hpp"


void read_buf( char *buf, unsigned max )
{
  char *p;

  p = buf;
  printf( "Input    > " );
  lenbuf = 0;
  if( fgets( buf, max, in ) == NULL )
  {
    fclose( in );
    in = stdin;
    fgets( buf, max, in );
  };
  lenbuf = strlen( buf );
  if( *(p + lenbuf - 1 ) == '\n' ) *(p + lenbuf - 1) = '\0';
  lenbuf = strlen( buf );
  while( buf[ lenbuf -1 ] == '\\' )
  {
    fgets( &buf[ ( lenbuf ? lenbuf - 1 : 0 ) ], max - lenbuf, in );
    lenbuf = strlen( buf );
    if( *(p + lenbuf - 1 ) == '\n' ) *(p + lenbuf - 1) = '\0';
    lenbuf = strlen( buf );
  };
  puts( buf );
}

char *skip_white( char *s )
{
  char *p, *q;

  p = s;
  while( isspace( *p ) ) p++;
  q = s;
  while( *q++ = *p++ );
  return s;
};

void main( int argc, char *argv[] )
{
  FILE *f;
  int i, j, k, arret;
  int c;
  int declare;
```

```
  char nom[MAXVARLEN+1], nom2[MAXVARLEN+1], nom3[MAXVARLEN+1], nom4[MA
XVARLEN+1], *p, *q;
  int bi, bs;
  ListeLineaire l, tmp1, tmp2, trad, *l1, *l2;
  ClasseExpression ee1, ee2;

  in = stdin;
  init_mem( );

  if( argc > 1 )
  {
    in = fopen( argv[1], "rt" );
  };

  declare = 1;
  setjmp( jumper );
  read_buf( buf, MAXBUF );
  skip_white( buf );

  while( strncmp( buf, "enddeclare", 10 ) )
  {
    p = buf;
    trad.detruire( );
    env.repositionner( );
    tmp2.detruire( );
    tmp1.detruire( );
    if( lenbuf )
    {
      if( strncmp( buf, "declare", 7 ) == 0 )
      { if( !declare ) term_mem0( );
        declare = 1;
        p += 7;
        p = whites( p );
        q = nom;
        i = 0;
        if( !( isalpha( *p ) && ( *p != '_' ) ) )
        {
          error( &trad, "identifier expected" );
        };
        while( ( i < MAXVARLEN ) && ( isalnum( *p ) || ( *p == '_' ) )
)
        {
          *q++ = *p++;
          i++;
        };
        *q = '\0';

        if( isalnum( *p ) || ( *p == '_' ) )
        {
          warning( "Identifier to long : truncate to %d character(s)",
MAXVARLEN );
          while( isalnum( *p ) || ( *p == '_' ) ) p++;
        };

        p = whites( p );

        o = dans_env( nom );
        if( o != NULL )
        {
          error( &trad, "Identifier '%s' already defined", nom );
        };

        if( strncmp( p, "as", 2 ) )
        {
```

```
          error( &trad, "%s\n'as' expected", buf );
        };
        p = whites( p + 2 );

        if( strncmp( p, "integer", 7 ) == 0 )
        {
          ve = allouer_var_ent( nom );
          p = whites( p + 7 );
          i = ENT_NON_DEF;
          if( *p )
          {
            if( !isdigit( *p ) && ( *p != '+' ) && ( *p != '-' ) ) err
or( &trad, "integer expected" );
            i = atoi( p );
            *ve = i;
          };
        }
        else if( strncmp( p, "boolean", 7 ) == 0 )
        {
          vl = allouer_var_log( nom );
          p = whites( p + 7 );
          i = ENT_NON_DEF;
          if( *p )
          {
            if( strncmp( p, "vrai", 4 ) && strncmp( p, "faux", 4 ) ) e
rror( &trad, "boolean value expected" );
            if( strncmp( p, "vrai", 4 ) == 0 ) i = 1;
            else if( strncmp( p, "faux", 4 ) == 0 ) i = 0;
          };
          *vl = i;
        }
        else if( strncmp( p, "array", 5 ) == 0 )
        {
          p = whites( p + 5 );
          if( strncmp( p, "from", 4 ) ) error( &trad, "%s\n'from' expe
cted", buf );

          p = whites( p + 4 );
          if( !isdigit( *p ) && ( *p != '+' ) && ( *p != '-' ) ) error
( &trad, "integer value expected" );

          bi = atoi( p );
          while( isdigit( *p ) || ( *p == '+' ) || ( *p == '-' ) ) p++
;
          p = whites( p );

          if( strncmp( p, "to", 2 ) ) error (&trad, "%s\n'to' expected
" );
          p = whites( p + 2 );

          if( !isdigit( *p ) && ( *p != '+' ) && ( *p != '-' ) ) error
( &trad, "integer value expected" );
          bs = atoi( p );
          while( isdigit( *p ) || ( *p == '+' ) || ( *p == '-' ) ) p++
;
          p = whites( p );

          t = allouer_tab( nom, bi, bs );

          if( *p )
          {
            for( i = bi; (i <= bs) && *p; i++ )
            {
              k = 0;
```

```
                j = 1;
                if( *p == '+' ) p++;
                if( *p == '-' )
                { j = -1;
                  p++;
                };
                while( isdigit( *p ) ) k = k * 10 + *p++ - '0';

                k = k * j;
                (*(ClasseTableau *)t)(i) = k;
                p = whites( p );
              };
              for( ; i <= bs; i++ ) (*(ClasseTableau *)t)(i) = ENT_NON_D
EF;
            };
          }
          else if( strncmp( p, "segment", 7 ) == 0 )
          {
            p = whites( p + 7 );
            if( strncmp( p, "from", 4 ) ) error( &trad, "%s\n'from' expe
cted", buf );

            p = whites( p + 4 );
            q = nom2;
            i = 0;
            if( !( isalpha( *p ) && ( *p != '_' ) ) )
            {
              error( &trad, "identifier expected" );
            };
            while( ( i < MAXVARLEN ) && ( isalnum( *p ) || ( *p == '_' )
 ) )
            {
              *q++ = *p++;
              i++;
            };
            *q = '\0';

            if( isalnum( *p ) || ( *p == '_' ) )
            {
              warning( "Identifier to long : truncate to %*.*s character
(s)", MAXVARLEN, MAXVARLEN, nom2 );
              while( isalnum( *p ) || ( *p == '_' ) ) p++;
            };

            p = whites( p );

            o2 = dans_env( nom2 );
            if( o2 == NULL )
            {
              error( &trad, "Identifier '%*.*s' unknow", MAXVARLEN, MAXV
ARLEN, nom2 );
            };

            if( strncmp( p, "between", 7 ) ) error( &trad, "%s\n'between
' expected", buf );
            p = whites( p + 7 );

            p = traduction( &tmp1, p, '&' );
            ee1 = ClasseExpression( &tmp1 );
            p = whites( p + 1 );

            p = traduction( &tmp2, p, '\0' );
            ee2 = ClasseExpression( &tmp2 );
```

```
          FF = allouer_seg( (ClasseTableau *)o2, nom, &ee1, &ee2 );

          tmp1.detruire( );
          tmp2.detruire( );
        }
        else if( strncmp( p, "expression", 10 ) == 0 )
        {
          p = whites( p + 10 );

          traduction( &trad, p, '\0' );

          e1 = new ClasseExpression ( &trad, nom );

          trad.detruire( );

          env.repositionner( );
          env.inserer( e1 );
        };
      }
      else if( strncmp( p, ";", 1 ) == 0 )
      {

      }
      else printf( "ERROR : %s\n\n", buf );
    };

    read_buf( buf, MAXBUF );
    skip_white( buf );
  };

  setjmp( jumper );

  read_buf( buf, MAXBUF );
  skip_white( buf );

  arret = 0;

  while( strncmp( buf, "endprog", 7 ) && !arret )
  {
    p = buf;
    trad.detruire( );
    env.repositionner( );
    tmp2.detruire( );
    tmp1.detruire( );
    if( lenbuf )
    {
      if( strncmp( p, "eval", 4 ) == 0 )
      {
        p = whites( p + 4 );
        q = nom;
        i = 0;
        if( !( isalpha( *p ) && ( *p != '_' ) ) )
        {
          error( &trad, "identifier expected" );
        };
        while( ( i < MAXVARLEN ) && ( isalnum( *p ) || ( *p == '_' ) )
)
        {
          *q++ = *p++;
          i++;
        };
        *q = '\0';

        if( isalnum( *p ) || ( *p == '_' ) )
```

```
        {
          warning( "Identifier to long : truncate to %d character(s)",
MAXVARLEN );
          while( isalnum( *p ) || ( *p == '_' ) ) p++;
        };

        p = whites( p );

        o = dans_env( nom );
        if( o == NULL )
        {
          error( &trad, "Unknow '%s'", nom );
        };

        arret = !ajo_inst_eval( o );
      }
      else if( strncmp( p, "let", 3 ) == 0 )
      {
        p = whites( p + 3 );

        if( !isalpha( *p ) && ( *p != '_' ) ) error( &trad, "'%s' : id
entifier expected", p );

        trad.detruire( );

        q = nom;
        i = 0;
        while( ( isalnum( *p ) || ( *p == '_' ) ) && ( i < MAXVARLEN )
)
        {
          *q++ = *p++;
          i++;
        };
        *q = '\0';

        while( isalpha( *p ) || ( *p == '_' ) ) p++;

        p = whites( p );

        o = dans_env( nom );

        if( !o ) error( &trad, "'%s' : unknow identifier", nom );

        if( *p == '[' )
        {
          p = traduction( &trad, p + 1, ']' );
          p++;

          e1 = new ClasseExpression( &trad, "" );

          p = whites( p + 1 );
          trad.detruire( );
        }
        else e1 = NULL;


        p = traduction( &trad, p, '\0' );
        trad.repositionner( );
        if( !trad.eof( ) )
        {
          e2 = new ClasseExpression( &trad );
          trad.detruire( );
        }
        else e2 = NULL;
```

```
      arret = !ajo_inst_let( o, e1, e2 );
    }
    else if( strncmp( p, "if", 2 ) == 0 )
    {
      p = whites( p + 2 );
      p = traduction( &trad, p, 'g' );
      if( strncmp( p, "goto", 4 ) ) error( &trad, "'goto' expected"
);
      p = whites( p + 4 );

      i = atoi( p );

      e1 = new ClasseExpression( &trad );
      arret = !ajo_inst_si( e1, i );

    }
    else if( strncmp( p, "goto", 4 ) == 0 )
    {
      p = whites( p + 4 );
      i = atoi( p );
      arret = !ajo_inst_goto( i );
    }
    else if( strncmp( p, "etiq", 4 ) == 0 )
    {
      p = whites( p + 4 );
      if( !isdigit( *p ) ) error( &trad, "number expected" );
      i = atoi( p );
      j = adr_eti( i );
      if( j != -1 ) error( &trad, "label already defined" );
      arret = !ajo_adr( i );
    }
    else if( strncmp( p, "stop", 4 ) == 0 )
    {
      arret = !ajo_inst_stop( );
    }
    else if( strncmp( p, "printvar", 8 ) == 0 )
    {
      p = whites( p + 8 );

      if( !isalpha( *p ) && ( *p != '_' ) ) error( &trad, "'%s' : id
entifier expected", p );

      trad.detruire( );

      q = nom;
      i = 0;
      while( ( isalnum( *p ) || ( *p == '_' ) ) && ( i < MAXVARLEN )
)
      {
        *q++ = *p++;
        i++;
      };
      *q = '\0';

      while( isalpha( *p ) || ( *p == '_' ) ) p++;

      p = whites( p );

      o = dans_env( nom );

      if( !o ) error( &trad, "'%s' : unknow identifier", nom );

      if( *p == '[' )
```

```
        {
          p = traduction( &trad, p + 1, ']' );
          p++;

          trad.repositionner( );
          if( !trad.eof( ) )
          {
            e1 = new ClasseExpression( &trad );
            trad.detruire( );
          }
          else e1 = NULL;
        }
        else e1 = NULL;

        arret = !ajo_inst_priv( o, e1 );
      }
      else if( strncmp( p, "printnl", 7 ) == 0 )
      {
        arret = !ajo_inst_nl( );
      }
      else if( strncmp( p, "print", 5  ) == 0 )
      {
        p = whites( p + 5 );
        q = p;
        p++;
        while( *p && ( *p != *q ) ) p++;
        *p = '\0';
        q++;
        arret = !ajo_inst_pri( q );
      }
      else if( strncmp( p, "read", 4 ) == 0 )
      {
        p = whites( p + 4 );

        if( !isalpha( *p ) && ( *p != '_' ) ) error( &trad, "'%s' : id
entifier expected", p );

        trad.detruire( );

        q = nom;
        i = 0;
        while( ( isalnum( *p ) || ( *p == '_' ) ) && ( i < MAXVARLEN )
)
        {
          *q++ = *p++;
          i++;
        };
        *q = '\0';

        while( isalpha( *p ) || ( *p == '_' ) ) p++;

        p = whites( p );

        o = dans_env( nom );

        if( !o ) error( &trad, "'%s' : unknow identifier", nom );

        if( *p == '[' )
        {
          p = traduction( &trad, p + 1, ']' );
          p++;

          trad.repositionner( );
          if( !trad.eof( ) )
```

```
        {
          e1 = new ClasseExpression( &trad );
          trad.detruire( );
        }
        else e1 = NULL;
      }
      else e1 = NULL;

      arret = !ajo_inst_read( o, e1 );
    }
    else if( strncmp( p, "flush", 5 ) == 0 )
    {
      arret = !ajo_inst_flush( );
    }
    else if( strncmp( p, "print", 5  ) == 0 )
    {
      p = whites( p + 5 );
      q = p;
      p++;
      while( *p && ( *p != *q ) ) p++;
      *p = '\0';
      q++;
      arret = !ajo_inst_pri( q );
    }
    else if( strncmp( p, ";", 1 ) == 0 )
    {
    }
    else printf( "ERROR : %s\n", buf );
  }

  read_buf( buf, MAXBUF );
  skip_white( buf );
};

setjmp( jumper );

read_buf( buf, MAXBUF );
skip_white( buf );

arret = 0;

while( strncmp( buf, "quit", 4 ) && !arret )
{
  p = buf;
  trad.detruire( );
  env.repositionner( );
  tmp2.detruire( );
  tmp1.detruire( );
  if( lenbuf )
  {
    if( strncmp( p, "run", 3 ) == 0 )
    {
      init_mem0( );
      p = whites( p + 3 );
      if( *p ) i = atoi( p );
      else i = -1;
      run_program( i );
      term_mem0( );
    }
    else if( strncmp( p, "ev", 2 ) == 0 )
    {
      p = whites( p + 2 );
      p = traduction( &trad, p, '\0' );
      ee1 = ClasseExpression( &trad );
```

```
        ee1.eval( );
        printf( "Result : %d\n", resultat( ) );
        trad.detruire( );
        (ee1.Liste( ) )->detruire( );
      }
      else if( strncmp( p, ";", 1 ) == 0 )
      {
      }
      else printf( "ERROR : %s\n", buf );
    }

    read_buf( buf, MAXBUF );
    skip_white( buf );
  };

  if( in != stdin )
  {
    fclose ( in );
  }

  term_mem( );

}
```

Fichier PRINC2.CPP

Fichier de traduction

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <stdarg.h>
#include <setjmp.h>
#include <stdlib.h>

#include "cconst.hpp"
#include "cbase.hpp"
#include "listelin.hpp"
#include "var.hpp"
#include "fct.hpp"
#include "expr.hpp"

#undef __PRINCIPAL__
#include "principal.hpp"

#define MAXPILE  20

static char *output_op[] = {"", "or", "and", "=", "<>", "<", "<=", ">"
,
                            ">=", "+", "-", "*", "/", "mod", "not", "(
)" };

int priorite( int p )
{
  if( p == 0 ) return 0;
  if( p == 1 ) return 4;
  if( p == 2 ) return 5;
  if( p == 3 ) return 9;
  if( p == 4 ) return 9;
  if( p == 5 ) return 10;
  if( p == 6 ) return 10;
  if( p == 7 ) return 10;
  if( p == 8 ) return 10;
  if( p == 9 ) return 12;
  if( p == 10 ) return 12;
  if( p == 11 ) return 13;
  if( p == 12 ) return 13;
  if( p == 13 ) return 13;
  if( p == 14 ) return 15;
  if( p == 15 ) return 16;
  return 0;
}

int code_op( char *p )
{
  if( strcmp( p, "(" ) == 0 ) return 0;
  if( strcmp( p, ")" ) == 0 ) return 0;
  if( strcmp( p, "or" ) == 0 ) return 1;
  if( strcmp( p, "OR" ) == 0 ) return 1;
  if( strcmp( p, "AND" ) == 0 ) return 2;
  if( strcmp( p, "and" ) == 0 ) return 2;
  if( strcmp( p, "=" ) == 0 ) return 3;
  if( strcmp( p, "<>" ) == 0 ) return 4;
  if( strcmp( p, "<" ) == 0 ) return 5;
  if( strcmp( p, "<=" ) == 0 ) return 6;
  if( strcmp( p, ">" ) == 0 ) return 7;
  if( strcmp( p, ">=" ) == 0 ) return 8;
  if( strcmp( p, "+" ) == 0 ) return 9;
  if( strcmp( p, "-" ) == 0 ) return 10;
  if( strcmp( p, "*" ) == 0 ) return 11;
  if( strcmp( p, "/" ) == 0 ) return 12;
  if( strcmp( p, "MOD" ) == 0 ) return 13;
```

```
  if( strcmp( p, "mod" ) == 0 ) return 13;
  if( strcmp( p, "NOT" ) == 0 ) return 14;
  if( strcmp( p, "not" ) == 0 ) return 14;
  if( strcmp( p, "()" ) == 0 ) return 15;
  return 0;
}

char *whites( char *p )
{
  while( isspace( *p ) ) p++;
  return p;
}

char *operateurs[] = {"=", "<>", "<=", "<", ">=", ">", "+", "-", "*",
"/",
                      "and", "AND", "or", "OR", "mod", "MOD",
                       0};

char *fonctions[] = { "not", "somme", "produit", "min", "max", "long",
                      "first", "last", "inchange", "empty",
                      "tricrois", "tridec", "tristrcrois",
                      "tristrdec", "pair", "examine", "permute", "vrai
",
                      "faux", "head", "tail", "egal", "permutation",
                      "segment", "prefixe", "suffixe", "concat",
                      "pourtout", "ilexiste", NULL };

int numfct[] = {_ClasseNot, _ClasseSomme, _ClasseProduit, _ClasseMin,
                _ClasseMax, _ClasseLong, _ClasseFirst, _ClasseLast,
                _ClasseInchange, _ClasseEmpty, _ClasseTriCroi, _Classe
TriDec,
                _ClasseTriStrCroi, _ClasseTriStrDec, _ClassePair,
                _ClasseExamine, _ClassePermute, 0, 0, _ClasseHead,
                _ClasseTail, _ClasseEgal, _ClassePermutation,
                _ClasseIsSegment, _ClassePrefixe, _ClasseSuffixe,
                _ClasseConcat, _ClassePourTout, _ClasseIlExiste };

char *fct_seg[] = {"HEAD", "head", "TAIL", "tail", 0};

char *get_nom( char *s, char *buf )
{
  while( isalnum( *s ) || ( *s == '_' ) ) *buf++ = *s++;
  *buf = '\0';
  return s;
}

ClasseConstante vrai( 1 ), faux( 0 );

void error( ListeLineaire *trad, char *s, ... )
{
  va_list argptr;
  ClasseBase *o;

  printf("\n\nError : ");
  va_start( argptr, s );
  vprintf( s, argptr );
  printf( "\n" );
  va_end( argptr );

  trad->detruire( );
  longjmp( jumper, 1 );
  exit( 1 );
}
```

```
void warning( char *s, ... )
{
  va_list argptr;

  printf("\n\nWarning : ");
  va_start( argptr, s );
  vprintf( s, argptr );
  printf( "\n" );
  va_end( argptr );
}

char *traduction( ListeLineaire *trad, char *s, char term )
{
  int h;
  int pile[MAXPILE];
  char *p, c, nom[33], *q, buf[51];
  int sign, pr;
  int lennom, i;
  long l;
  ClasseBase *o;
  ListeLineaire tmp, tmp1, tmp2;
  ClasseVarEntiere *tve;


  for( i = 0; i < MAXPILE; i++ ) pile[ i ] = 0;
  p = s;
  h = 0;

A:
  p = whites( p );
  c = *p;
  sign = 0;
  if( c == '+' )
  { sign = 0;
    p++;
    p = whites( p );
    c = *p;
  }
  else
  if( c == '-' )
  {
    sign = 1;
    p++;
    p = whites( p );
    c = *p;
  };

  if( isalpha( c ) || ( c == '_' ) )
  {
    lennom = 0;
    nom[lennom++] = *p++;
    while( *p && ( lennom < 32 ) && isalnum( *p ) )
    {
      nom[ lennom++ ] = *p++;
    };
    nom[ lennom ] = '\0';

    if( isalnum( *p ) || ( *p == '_' ) ) warning( "%s%s : identifier t
o long", nom, p );

    while( isalnum( *p ) || ( *p == '_' ) ) p++;
    p = whites( p );

    q = fonctions[0];
```

```
i = 0;
while( q && strcmp( nom, q ) )
{ i++;
  q = fonctions[i];
};
if( q )
{
  if( i == 0 ) /* Not */
  {
    strcpy( buf, nom );
    q = p;
    goto D;
  }
  else if( ( i >=  1 ) && ( i <= 16 ) )
  {
    p = whites( p );
    if( *p != '(' ) error( trad, "%s\n'(' expected", p );
    p = whites( p + 1 );
    p = traduction( &tmp, p, ')' );

    tmp.repositionner( );
    if( !tmp.eof( ) ) o = tmp.courant( );
    else o = NULL;
    tmp.avancer( );
    while( !tmp.eof( ) )
    {
      trad->inserer( o );
      trad->avancer( );
      o = tmp.courant( );
      tmp.avancer( );
    };
    tmp.detruire( );

    p = whites( p );
    if( *p != ')' ) error( trad, "')' missing" );

    p++;

    f = new ClasseFonction( numfct[i] );
    trad->inserer( f );
    trad->avancer( );

    trad->inserer( o );
    trad->avancer( );

    strcpy( buf, "" );

  }
  else if( i == 17 )
  {
    trad->inserer( &vrai );
    trad->avancer( );
  }
  else if( i == 18 )
  {
    trad->inserer( &faux );
    trad->avancer( );
  }
  else if( ( i >= 19 ) && ( i <= 20 ) )
  {
    p = whites( p );
    if( *p != '(' ) error( trad, "%s\n'(' expected", p );
    p = whites( p + 1 );
    p = traduction( &tmp, p, ')' );
```

```
        tmp.repositionner( );
        if( !tmp.eof( ) ) o = tmp.courant( );
        else o = NULL;
        tmp.avancer( );
        while( !tmp.eof( ) )
        {
          trad->inserer( o );
          trad->avancer( );
          o = tmp.courant( );
          tmp.avancer( );
        };
        tmp.detruire( );

        f = new ClasseFonction( numfct[i] );
        trad->inserer( f );
        trad->avancer( );

        if( ( o->isA( ) != _ClasseTableau ) && (o->isA( ) != _ClasseSe
gment ) )
                error( trad, "Array or segment identifier expected" );

        sprintf( buf, "__TEMP%d", NextSegment++ );
        FF = new ClasseSegment(  );

        trad->inserer( o );
        trad->avancer( );

        trad->inserer( FF );
        trad->avancer( );

      }
      else if( ( i >= 21 ) && ( i <= 25 ) )
      {
        p = whites( p );
        if( *p != '(' ) error( trad, "%s\n'(' expected", p );
        p = whites( p + 1 );
        p = traduction( &tmp, p, ',' );

        tmp.repositionner( );
        if( !tmp.eof( ) ) o = tmp.courant( );
        else o = NULL;
        tmp.avancer( );
        while( !tmp.eof( ) )
        {
          trad->inserer( o );
          trad->avancer( );
          o = tmp.courant( );
          tmp.avancer( );
        };
        tmp.detruire( );

        o1 = o;

        p = whites( p + 1 );
        p = traduction( &tmp, p, ')' );

        tmp.repositionner( );
        if( !tmp.eof( ) ) o = tmp.courant( );
        else o = NULL;
        tmp.avancer( );
        while( !tmp.eof( ) )
        {
          trad->inserer( o );
```

```
      trad->avancer( );
      o = tmp.courant( );
      tmp.avancer( );
    };
    tmp.detruire( );

    p = whites( p );
    if( *p != ')' ) error( trad, "')' missing" );

    p++;

    f = new ClasseFonction( numfct[i] );
    trad->inserer( f );
    trad->avancer( );

    trad->inserer( o1 );
    trad->avancer( );

    trad->inserer( o );
    trad->avancer( );

    strcpy( buf, "" );

  }
  else if( i == 26 )
  {
    p = whites( p );
    if( *p != '(' ) error( trad, "%s\n'(' expected", p );
    p = whites( p + 1 );
    p = traduction( &tmp, p, ',' );

    tmp.repositionner( );
    if( !tmp.eof( ) ) o = tmp.courant( );
    else o = NULL;
    tmp.avancer( );
    while( !tmp.eof( ) )
    {
      trad->inserer( o );
      trad->avancer( );
      o = tmp.courant( );
      tmp.avancer( );
    };

    o1 = o;

    tmp.detruire( );

    p = whites( p + 1 );
    p = traduction( &tmp, p, ',' );

    tmp.repositionner( );
    if( !tmp.eof( ) ) o = tmp.courant( );
    else o = NULL;
    tmp.avancer( );
    while( !tmp.eof( ) )
    {
      trad->inserer( o );
      trad->avancer( );
      o = tmp.courant( );
      tmp.avancer( );
    };
    o2 = o;
    tmp.detruire( );
```

```
        p = whites( p + 1 );
        p = traduction( &tmp, p, ')' );

        tmp.repositionner( );
        if( !tmp.eof( ) ) o = tmp.courant( );
        else o = NULL;
        tmp.avancer( );
        while( !tmp.eof( ) )
        {
          trad->inserer( o );
          trad->avancer( );
          o = tmp.courant( );
          tmp.avancer( );
        };
        tmp.detruire( );

        p = whites( p );
        if( *p != ')' ) error( trad, "')' missing" );

        p++;

        f = new ClasseFonction( numfct[i] );
        trad->inserer( f );
        trad->avancer( );

        trad->inserer( o1 );
        trad->avancer( );

        trad->inserer( o2 );
        trad->avancer( );

        trad->inserer( o );
        trad->avancer( );

        strcpy( buf, "" );
      }
      else if( ( i == 27 ) || ( i == 28 ) ) /* pourtout <nom> <exp> :
<exp> : <exp>
                                              ilexiste <nom> <exp> :
<exp> : <exp> */
      {
        p = whites( p );
        if( isalpha( c ) || ( c == '_' ) )
        {
          lennom = 0;
          nom[lennom++] = *p++;
          while( *p && ( lennom < 32 ) && isalnum( *p ) )
          {
            nom[ lennom++ ] = *p++;
          };
          nom[ lennom ] = '\0';
          while( isalnum( *p ) ) p++;

          o = dans_env( nom );
          if( o == NULL ) error( trad, "'%s' : unknow variable", nom )
;
          tve = (ClasseVarEntiere *)o;
          p = whites( p );
        }
        else error( trad, "identifier expected : %s", buf );

        p = traduction( &tmp, p, ':' );

        p = whites( p + 1 );
```

```
        p = traduction( &tmp1, p, ':' );

        p = whites( p + 1 );
        p = traduction( &tmp2, p, term );

        if( i == 27 ) o = (ClasseBase *)new ClassePourTout( tve, new C
lasseExpression( &tmp ),
                                    new ClasseExpression( &tmp1 ),
                                    new ClasseExpression( &tmp2 ) );
        else (ClasseBase *)new ClasseIlExiste( tve, new ClasseExpressi
on( &tmp ),
                                    new ClasseExpression( &tmp1 ),
                                    new ClasseExpression( &tmp2 ) );
        trad->inserer( o );
        trad->avancer( );
        tmp.detruire( );
        tmp1.detruire( );
        tmp2.detruire( );

        strcpy( buf, "" );
      }
      else
      {
        error( trad, "'%s' : unknow", nom );
      };
    }
    else
    {
      o = dans_env( nom );

      if( !o ) error( trad, "'%s' : unknow identifier", nom );

      p = whites( p );

      i = o->isA( );
      if( ( i == _ClasseTableau ) || ( i == _ClasseSegment ) )
      {
        if( *p == '[' )
        {
          p = traduction( trad, p+1, ']' );
          p++;
          o = (ClasseBase *)new ClasseVarInd( (ClasseTabSeg *)o );
        }
        else if( *p == '{' )
        {
          p = traduction( &tmp, p+1, ':' );
          p++;
          p = traduction( &tmp1, p, '}' );
          p++;
          sprintf( buf, "__TEMP%d", NextSegment++ );
          e1 = new ClasseExpression( &tmp );
          e2 = new ClasseExpression( &tmp1 );
          FF = new ClasseSegment( (ClasseTableau *)o, buf, e1, e2 );
          tmp.detruire( );
          tmp1.detruire( );
          o = (ClasseBase *)FF;
        };
      };
        trad->inserer( o );
        trad->avancer( );


    };
  }
```

```
  else
  {
    if( ('0' <= c) && (c <= '9') )
    {
      l = 0l;
      while ( isdigit( *p ) )
      {
        l = l * 10 + *p - '0';
        p++;
      };
      if( l > (long)MAXVAL ) error( trad, "%l : number to big", l );
      i = (int)l;
      cc = new ClasseConstante( i );
      trad->inserer( cc );
      trad->avancer( );
    }
    else
    if( c == '(' )
    {
      p++;
      p = traduction( trad, p, ')' );
      if ( *p != ')' ) error( trad, "')' missing" );
      p++;
    };
  };
  if( sign )
  {
    f = (ClasseFonction *)new ClasseFonction( _ClasseNeg );
    trad->inserer( f );
    trad->avancer( );
  };
  p = whites( p );
  i = 0;
  q = operateurs[0];
  while( q && ( strstr( p, q ) != p ) ) q = operateurs[++i];
  if( q )
  { strcpy( buf, q );
    q = p + strlen( buf );
  }
  else
  { if( *p != term ) error( trad, "\"%s\" : unknow operator", p );
    strcpy( buf, "" );
  };

D:
  pr = priorite( code_op( buf ) );
  if( pr ) p = q;

B:
  if( h == 0 ) goto C;
  if( priorite( pile[ h-1 ] ) < pr ) goto C;

  switch( pile[ --h ] ) {
  case 1: /* or */
        f = new ClasseFonction( _ClasseOr );
        break;
  case 2: /* and */
        f = new ClasseFonction( _ClasseAnd );
        break;
  case 3: /* = */
        f = new ClasseFonction( _ClasseEq );
        break;
  case 4: /* <> */
        f = new ClasseFonction( _ClasseNe );
```

```
        break;
    case 5: /* < */
        f = new ClasseFonction( _ClasseLt );
        break;
    case 6: /* <= */
        f = new ClasseFonction( _ClasseLe );
        break;
    case 7: /* > */
        f = new ClasseFonction( _ClasseGt );
        break;
    case 8: /* >= */
        f = new ClasseFonction( _ClasseGe );
        break;
    case 9: /* + */
        f = new ClasseFonction( _ClassePlus );
        break;
    case 10: /* - */
        f = new ClasseFonction( _ClasseMoins );
        break;
    case 11: /* * */
        f = new ClasseFonction( _ClasseFois );
        break;
    case 12: /* / */
        f = new ClasseFonction( _ClasseDiv );
        break;
    case 13: /* Mod */
        f = new ClasseFonction( _ClasseMod );
        break;
    case 14: /* Not */
        f = new ClasseFonction( _ClasseNot );
        break;
    };
    trad->inserer( f );
    trad->avancer( );

    pile[ h ] = 0;
    goto B;

C:
    if ( pr != 0 )
    {
      if( h != MAXPILE )
      {
        pile[h++] = code_op( buf );
        goto A;
      }
      else
         error( trad, "Stack overflow" );
    }
E:

    return p;
}
```